
ASCET V5.2

Reference Guide

Copyright

The data in this document may not be altered or amended without special notification from ETAS GmbH. ETAS GmbH undertakes no further obligation in relation to this document. The software described in it can only be used if the customer is in possession of a general license agreement or single license. Using and copying is only allowed in concurrence with the specifications stipulated in the contract.

Under no circumstances may any part of this document be copied, reproduced, transmitted, stored in a retrieval system or translated into another language without the express written permission of ETAS GmbH.

© **Copyright 2007** ETAS GmbH, Stuttgart

The names and designations used in this document are trademarks or brands belonging to the respective owners.

The name INTECRIO is a registered trademark of ETAS GmbH.

Document EC010005 R5.2.2 EN

Contents

The Modeling Language

1	Projects	13
1.1	The Task Schedule for the Operating System	13
1.1.1	Scheduling	15
1.1.2	Tasks	17
1.1.3	Processes	19
1.1.4	Application Modes	19
1.2	Modules and Processes	19
1.3	Interprocess Communication	20
2	Components	23
2.1	Modules vs. Classes	24
2.2	Definition and Instantiation of Components	25
2.3	The Interface of Components	27
2.3.1	The Interface of Classes	27
2.3.2	The Interface of Modules	29
2.4	Reusing Components	29
2.4.1	Hierarchical Class Structure	31
2.4.2	Hierarchical Module Structure	31
2.5	State Machines	32

2.5.1	State Machine Components	34
2.5.2	Semantics of State Machines	47
2.5.3	Semantics: Simple State Machines	48
2.5.4	Semantics: Junctions in State Machines	51
2.5.5	Semantics: Hierarchical State Machines.	57
2.5.6	Semantics: Summary	68
2.5.7	Simple Code Example	72
2.5.8	Optimizing the State Machine	73
2.5.9	State Machines as Classes.	83
3	Types and Elements	89
3.1	Basic Model Types	90
3.1.1	Scalar Types	90
3.1.2	Composite Types	91
3.1.3	Real-time Language Constructs	94
3.1.4	Special Types	95
3.2	The Kind of Elements	96
3.3	The Scope of Elements	99
3.4	User-defined Model Types	100
4	Data and Implementations	101
4.1	Data	101
4.2	Implementations	103
4.2.1	Implementations for Scalar Types	103
4.2.2	The Implementation of Composite Types	105
4.2.3	The Implementation of User-Defined Types.	105
4.2.4	Implementation Casts.	106
4.3	Code Generation with Implementations	108
4.3.1	Transformation of Data under Implementation	109
4.3.2	General Rules for the Implementation Transformation	109
4.4	The Implementation of Methods and Processes	110
5	Body Specification in ESDL	111
5.1	ESDL as a Modelling Language	111
5.2	Basic Elements	112
5.2.1	Working with Methods and Processes	112
5.2.2	ESDL Syntax	114
5.2.3	Variable Names.	114
5.2.4	Data Types	115
5.2.5	Type Conversion	115
5.2.6	Primitive Methods.	116

5.2.7	Literals and Constants	116
5.2.8	Comments	116
5.2.9	Operators	117
5.3	Implementation Casts in ESDL	119
5.4	Control Flow	120
5.4.1	If...Else	120
5.4.2	Switch...Case...Default	121
5.4.3	While	123
5.4.4	For	123
5.4.5	Break	124
5.5	Methods	124
5.5.1	This	126
5.5.2	Access Control	126
5.5.3	Direct Access Methods	127
5.6	Composite Data Types	127
5.6.1	Arrays	127
5.6.2	Matrices	129
5.6.3	One-dimensional Tables	129
5.6.4	Two-dimensional Tables	131
5.6.5	Distributions and Group Tables	133
5.7	Structures	134
5.8	Messages	135
5.9	Resources	136
5.10	Mathematical Functions	136
5.11	Accessing Block Diagrams from ESDL	138
5.12	Using ESDL in State Machines	139
5.13	Overview: ESDL Features Compared	141
6	Body Specification with Block Diagrams	143
6.1	Graphical Description of Elements	143
6.1.1	Basic Elements	144
6.1.2	Elements of User-defined Type	149
6.2	Expressions	149
6.2.1	Arithmetic Operators	151
6.2.2	Comparison Operators	151
6.2.3	Logical Operators	151
6.2.4	Conditional Operators	152
6.2.5	Other Operators	153
6.3	Statements	154
6.3.1	Assignment	155

6.3.2	The Break Statement	155
6.3.3	Method Call	156
6.3.4	Control Flow	156
6.4	The Semantics of Block Diagrams	159
6.4.1	Graphical Hierarchies	160
7	Body Specification in C	161
7.1	Structure	161
7.1.1	Methods and Processes	162
7.1.2	Variables and Function Parameters	163
7.1.3	Header	169
7.2	External Source Code	169
7.3	Programming Model Interface	170
7.4	Access Macros	170
8	Continuous Time Systems.	173
8.1	Structure of Continuous Time Models	173
8.1.1	Modeling with Basic Blocks and Structure Blocks	174
8.1.2	Modeling with Graphical Hierarchies	175
8.1.3	Experiments	176
8.1.4	Projects and Hybrid Projects	176
8.2	Solving Differential Equations – Integration Algorithms	177
8.2.1	Integration Methods – Overview.	179
9	Continuous Time Basic Blocks.	183
9.1	Basics	183
9.2	Available Elements and Methods	183
9.2.1	Modeling With Continuous Time Basic Blocks.	184
9.3	Block Interfaces	185
9.4	Block Methods	186
9.5	Computing Sequence	187
9.6	Modeling with ESDL	191
9.6.1	Differential Equations in ESDL.	191
9.6.2	Semantic Checks in ESDL	192
9.6.3	Additional Library Functions	193
9.7	Modeling in C	195
9.7.1	Differential Equations in C	195
9.7.2	Additional C Routines.	196
10	Continuous Time Structure Blocks and Graphical Hierarchies	199
10.1	Reuse of Structure Blocks	199
10.2	Elements of a Continuous Time Structure Block	199

10.3	Block Interfaces	200
10.4	Operators	200
10.5	Algebraic Loops	201
10.6	Direct and Nondirect Output	201
10.7	Difference Between Graphical Hierarchies and CT Structure Blocks	204
10.8	Computing Sequence of Methods Within a Structure	204
11	Projects and Hybrid Projects	209
11.1	Combining Continuous Time Blocks With Modules	210

Reference Lists

12	The ASCET System Library	215
12.1	Bit Operators	215
12.1.1	and	215
12.1.2	clearBit	215
12.1.3	getBit	216
12.1.4	or	216
12.1.5	rotate	217
12.1.6	setBit	217
12.1.7	shiftLeft	218
12.1.8	shiftRight	218
12.1.9	toggleBit	219
12.1.10	writeBit	219
12.1.11	writeByte	220
12.1.12	xor	220
12.2	Comparators	221
12.2.1	ClosedInterval	221
12.2.2	LeftOpenInterval	221
12.2.3	OpenInterval	222
12.2.4	RightOpenInterval	222
12.2.5	GreaterZero	223
12.3	Counter & Timer	223
12.3.1	CountDown	223
12.3.2	CountDownEnabled	224
12.3.3	Counter	224
12.3.4	CounterEnabled	225
12.3.5	StopWatch	225
12.3.6	StopWatchEnabled	226
12.3.7	Timer	226
12.3.8	TimerEnabled	227

12.3.9	TimerRetrigger.....	227
12.3.10	TimerRetriggerEnabled.....	228
12.4	Delay.....	228
12.4.1	DelaySignal.....	228
12.4.2	DelaySignalEnabled.....	229
12.4.3	DelayValue.....	229
12.4.4	DelayValueEnabled.....	230
12.4.5	TurnOffDelay.....	230
12.4.6	TurnOffDelayVariable.....	231
12.4.7	TurnOnDelay.....	232
12.4.8	TurnOnDelayVariable.....	232
12.5	Memory.....	233
12.5.1	Accumulator.....	233
12.5.2	AccumulatorEnabled.....	234
12.5.3	AccumulatorLimited.....	235
12.5.4	RSFlipFlop.....	235
12.6	Miscellaneous.....	236
12.6.1	DeltaOneStep.....	236
12.6.2	DifferenceQuotient.....	236
12.6.3	EdgeBi.....	237
12.6.4	EdgeFalling.....	237
12.6.5	EdgeRising.....	238
12.6.6	Muxlof4.....	238
12.6.7	Muxlof8.....	239
12.7	Nonlinears.....	239
12.7.1	Hysteresis-Delta-RSP.....	239
12.7.2	Hysteresis-LSP-Delta.....	240
12.7.3	Hysteresis-LSP-RSP.....	240
12.7.4	Hysteresis-MSP-DeltaHalf.....	241
12.7.5	Limiter.....	241
12.7.6	Signum.....	242
12.8	Transfer Function.....	242
12.8.1	Control.....	242
12.8.2	Integrators.....	248
12.8.3	Lowpass.....	252
13	Troubleshooting.....	257
13.1	General Hints.....	257
13.2	Problems with ASCET.....	258

14	Code Generation Messages	261
14.1	Components	261
14.1.1	Error Messages	261
14.1.2	Warnings	264
14.2	Projects	265
14.2.1	Error Messages	265
14.2.2	Warnings	266
14.3	Fixed Point Code Generation	266
14.3.1	Error Messages	266
14.3.2	Warnings	267
	Index	269

ASCET V5.2

The Modeling Language

1

Projects

In ASCET, an embedded software system is defined in the context of a project. A project contains at least the following:

- A collection of modules
- The task schedule for the real-time operating system
- The definition of the inter-process communication

The central part of a project is the definition of the operating system's task schedule. Here, the dynamic behavior of the system is described. Fig. 1-1 illustrates the structure of a project.

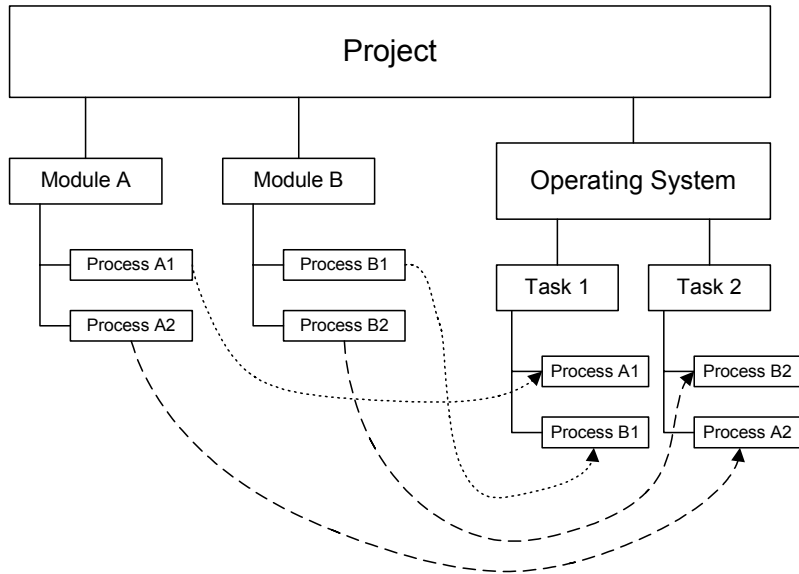


Fig. 1-1 The structure of a project

1.1

The Task Schedule for the Operating System

An essential part of an embedded control system is the underlying real-time operating system that controls the execution of the various algorithms and computations. In ASCET, the specification of the task schedule is supported by a special editor, where all relevant data for the operating system scheduling can be specified.

The specification of the task schedule is based on the automotive real-time operating system ERICOS^{EK}. To serve the large number of parallel requests to the embedded control system, e.g. camshaft interrupts or sampling at a fixed rate, a priority-based cooperative and preemptive scheduling is the core of the operating system. This scheduling controls the execution of tasks in a multi-tasking environment. A task is defined as a list of processes to be executed in a given order. A process is any portion of a control algorithm which has to be executed at a given rate or as a reaction to an external interrupt.

Since a control system contains a number of algorithms, the number of processes can be very large. At the same time, many of these processes have a similar dynamic behavior. The collection of processes with the same dynamic behavior into tasks therefore reduces the administrative overhead of the operating system and structures the dynamic behavior of the application. Processes with the same dynamic behavior are therefore collected into one task.

The definition of a real-time task schedule consists of:

- Scheduling
- Tasks
- Processes
- Application modes

1.1.1 Scheduling

The operating system schedules the execution of processes defined in the modules. The definition of the schedule consists of grouping processes into sequences where each sequence defines a task in the operating system task schedule. The tasks are activated by the operating system in different modes, for instance periodically by timers, or by software or external events.

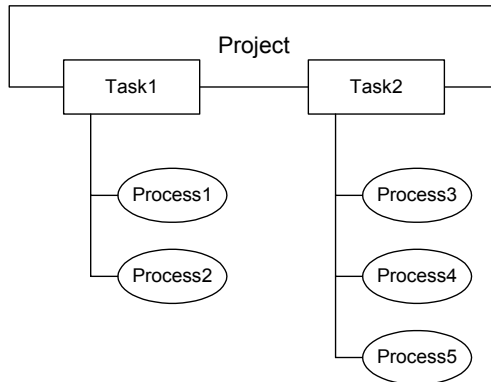


Fig. 1-2 Grouping processes into tasks

Fig. 1-2 shows two tasks with processes assigned to them. Task1 is activated every 10ms, and has a higher priority than Task2, which is activated every 20ms. The running times of the processes are as follows: $p_1 = 2\text{ms}$, $p_2 = 1\text{ms}$, $p_3 = 2\text{ms}$, $p_4 = 1\text{ms}$, $p_5 = 1\text{ms}$. The scheduling would then look like this:

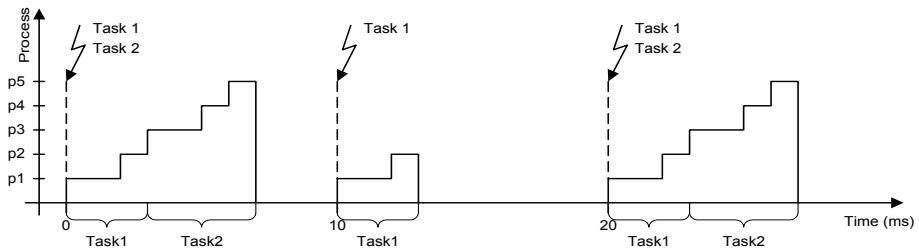


Fig. 1-3 A simple task schedule

The operating system knows three kinds of scheduling. In *cooperative scheduling*, the current process is not interrupted if a task with a higher priority is activated. A new task starts after the current process is finished. If the current task (the one that gets interrupted) has more processes to execute, it pauses until the interrupting task is completed. After the interrupting task is com-

pleted, the interrupted task is continued. This type of scheduling is illustrated in Fig. 1-4, where the running times of processes are $p_1 = 2\text{ms}$, $p_2 = 1\text{ms}$, $p_3 = 5\text{ms}$, $p_4 = 4\text{ms}$, and $p_5 = 2\text{ms}$.

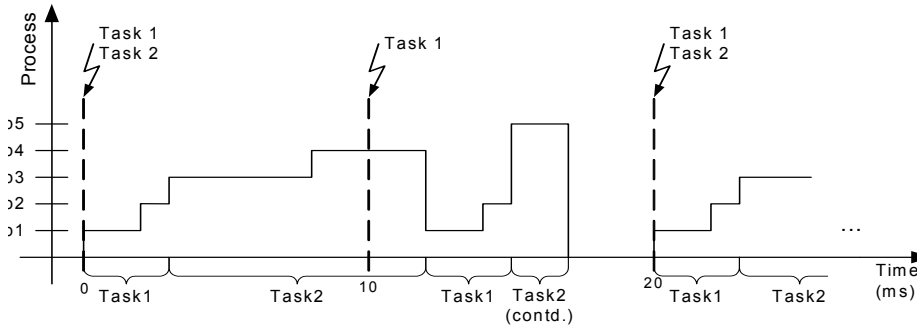


Fig. 1-4 Resuming an interrupted task in cooperative scheduling

In *preemptive scheduling* the current process is directly interrupted, whenever a task with a higher priority is activated. Since all cooperative tasks have lower priorities than preemptive, or non-preemptable, tasks (see Fig. 1-7), preemptive tasks cannot be interrupted by cooperative tasks. After the interrupting task is completed, the process is resumed. Fig. 1-5 shows the same scenario as above (i.e. the same process running times) with preemptive scheduling.

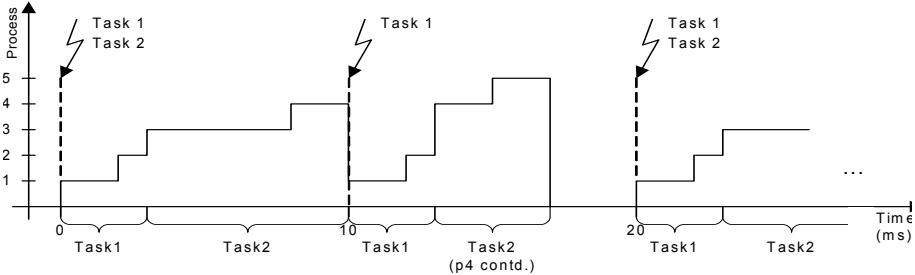


Fig. 1-5 Resuming an interrupted task in preemptive scheduling

In *non-preemptable scheduling* (micro-controller targets only), neither the current process nor the current task are interrupted when a task with higher priority is activated. The new task is activated only after the non-preemptable task is completed. Fig. 1-6 shows the same scenario as above (i.e. the same process running times) with non-preemptable scheduling

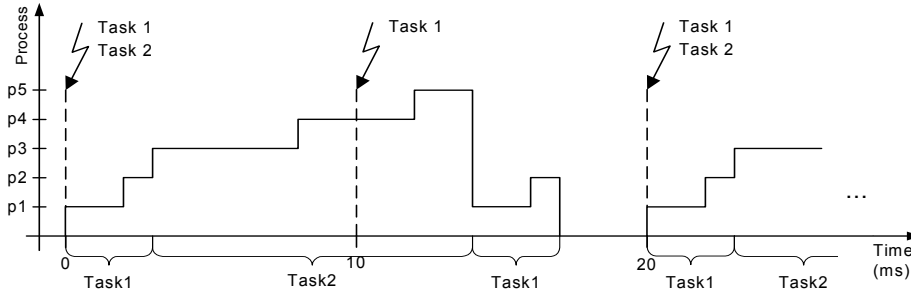


Fig. 1-6 Task-schedule for a non-preemptable task

Note

Non-preemptable tasks were introduced in ERCOS^{EK} to provide OSEK compatibility. Their use is explicitly not recommended, since most jobs can be solved easier by exclusive use of cooperative and preemptive tasks, and the configuration of non-preemptable tasks can be complicated due to several boundary conditions.

1.1.2 Tasks

A task contains a list of processes that are executed on activation of that task. The execution order of the processes is fixed. The way a task is scheduled by the scheduler of the operating system is defined by the task settings. There are several different task modes:

- *Alarm* tasks are activated periodically. The activation rate is specified in seconds.
- *Timetable* tasks (micro-controller targets only) are alarm tasks written into a timetable. Thus, runtime can be saved (at the price of enhanced memory requirement).
- *Interrupt* tasks are activated by an external event. For each processor, different types of events are available. The appropriate event can be chosen from a list of events.
- *Software* tasks are activated by calling an operating system routine, i.e. they are activated directly through the software.

- *Init* tasks are activated once before the start of the operation system. Init tasks contain code for the initialization of the system.

Each task is furthermore assigned to one of the three scheduling groups, non-preemptable, preemptive or cooperative, and inside each group to one of the available priority levels. The number of priority levels for each scheduling group can be defined by the user, and determines the memory demand of the scheduler tables. It should be optimized for the final system.

Tasks at a higher priority than the running task can interrupt the running task, running tasks scheduled as non-preemptable excepted. If the interrupting task belongs to the preemptive scheduling group, the running task is interrupted immediately, otherwise the interrupt happens at the end of the current process. Preemptive and non-preemptable tasks always have a higher priority than cooperative tasks. Fig. 1-7 shows the priority scheme. The actually available tasks depend on the selected target.

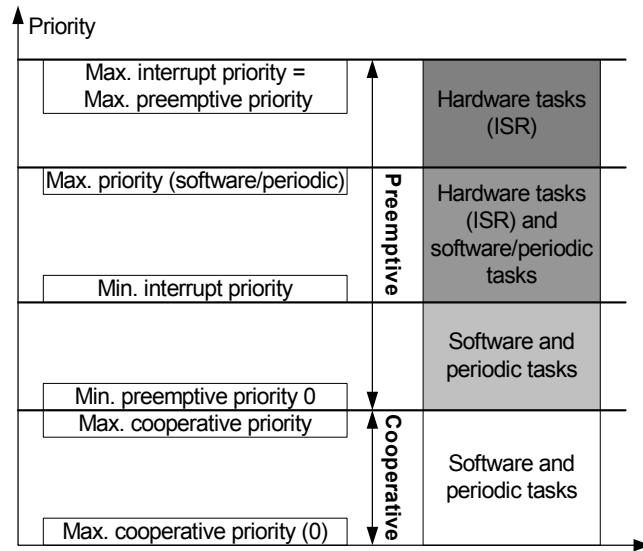


Fig. 1-7 Priority scheme

Each time a task is activated, the time elapsed since the previous activation is stored in the global variable ΔT . This variable can be used in the definition of algorithms to describe the control algorithms independent of their sample rate.

1.1.3 Processes

A task consists of a sequence of *processes*. Processes contain the execution code of the program. The body of a process is executed sequentially. Since tasks can be interrupted preemptively by tasks of a higher priority, processes can be interrupted in the middle of their execution. Therefore, processes must be designed so that they can be executed in parallel.

When working in a preemptive system, the main problem is data consistency. The operating system has to guarantee, that the result of the computation in a process depends on the value of the input variables alone, and not on the order of execution in the system.

To solve this problem, the ERCOS^{EK} concept of *messages* is supported in processes. In the ERCOS^{EK} operating system, messages are protected global variables. Protection is achieved by working on copies of the global variables. The system analyses whether a copy is required and establishes an optimum data consistency scheme without penalties for the run-time kernel.

1.1.4 Application Modes

Application Modes are a special feature of the operating system ERCOS^{EK}. In order to keep the run-time load of the processor low, the operating system can be run in different modes. Typical modes are the normal mode, the EEPROM programming mode etc. These modes are mutually exclusive, i.e. only one mode is active at a given time. Therefore, in each mode, only the relevant tasks have to be executed.

Each task is assigned an application mode which it runs in. Switches between application modes are activated by the software. When entering a new application mode, the init tasks assigned to that application mode are activated.

Note

Switches between application modes take place via an operating system service call. Details can be found in the API description of the ERCOS^{EK} manual.

1.2 Modules and Processes

The processes assigned to tasks are defined in the context of modules. A module encapsulates a number of related processes, e.g. processes that belong to a lambda control function. The functionality described in a module can be split into several processes, since different parts of a control algorithm may be computed at different times. This greatly reduces the execution time for the control algorithms, since only the most sensitive parts of the algorithms need to be

computed at the highest frequency. At the same time the descriptions of the algorithms are not distributed, which makes them easier to develop, maintain and understand.

The functionality of a complex control task can be distributed over several modules which can be modelled hierarchically. For further refinement classes and state machines can be used for sub-algorithms or service routines (e.g. accumulator, pi-control etc.)

Modules are exclusively used by projects and are the top level components within a project. Usually, modules are used to describe a unique part of a project, e.g. a lambda control. Therefore modules can have only one instance inside a project, in contrast to other components, which can have any number of instances (e.g. accumulators).

Like all other components, modules have an interface. The interface of a module consists of its processes and the messages which are used for data exchange.

1.3 Interprocess Communication

The communication between processes is achieved via messages, which are protected global variables in ERCOS^{EK}. Data consistency is achieved by working on copies of the actual variable whenever a copy is required.

Fig. 1-8 shows how data inconsistency may occur in a preemptive system. To avoid this conflict, the interprocess communication is modelled with messages. At the beginning of a process, all input messages (those messages that are only read) are received by the process. Upon receiving a message, an automatic temporary copy of the message is produced, on which the process works. At the end of the process, all messages that are written to are copied back to the actual message. This mechanism guarantees that the values of the variables are left unchanged within a process, unless the process itself changes its value.

The use of protected global variables for interprocess communication, i.e. the use of state messages, is appropriate for embedded control systems. There is no dependence between the sender and the receiver of a message, so that no complicated and run time consuming synchronization scheme is required. Secondly, when using state messages there is no one-to-one relation between a sender and the receiver. Therefore a message can be received by more than one process.

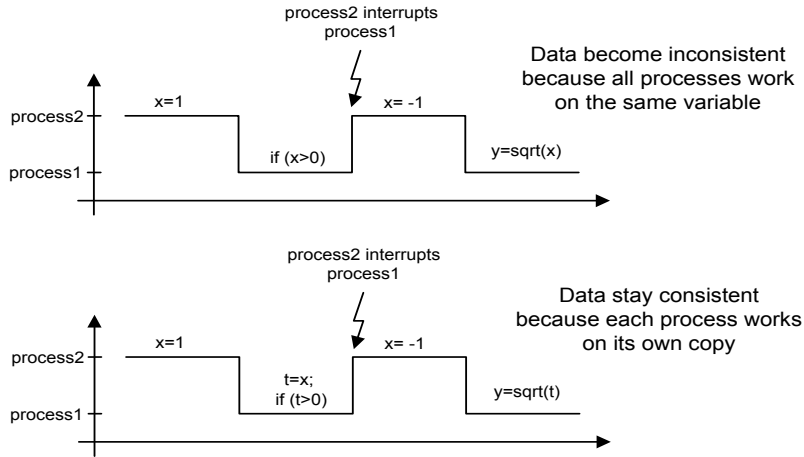


Fig. 1-8 Data inconsistency in a preemptive system

The messages mechanism is based on the ERCOS^{EK} message principle. The ERCOS^{EK} development environment contains an offline system optimization feature, where message implementation can be optimized. Here copies are only introduced, if data consistency is endangered, and copies are only produced at the beginning and the end of a task.

The interprocess communication is resolved by the project. Messages with the same name are bound to each other and represent the same message. If, for example, two processes use the message `velocity`, they communicate by writing to and reading from this variable. The same name-based resolution mechanism is performed on other global objects as well, e.g. global variables or global parameters.

2

Components

A project is at the top level of an embedded control system specification in ASCET. Here the framework of an application is defined and its execution are controlled. A project is the brain of an embedded control system.

Compared to this, components are the body. They are used to specify the actual control algorithms and other various computation tasks to be performed in the embedded control system.

Components have a clearly defined interface that describes how and when to perform the algorithms described in the components, and also how data exchange with other components is to be performed.

There are two types of components: modules and classes. A central aspect in the design of both types is data encapsulation, where ASCET follows an object-oriented approach. A component contains a number of elements that can be used by all processes or methods defined in that module or class. The scope of these elements can be restricted to be local. Even for messages, the scope can be restricted to processes defined within that module only.

A component specification consists of:

- The content of the component, i.e. declarations of the variables, parameters etc. the component uses.
- The interface of the component in the form of processes or methods. This interface can be extended by allowing access to internal variables (of classes) and messages (used in modules) directly.
- The algorithms themselves, which specify the computations within a process or method.

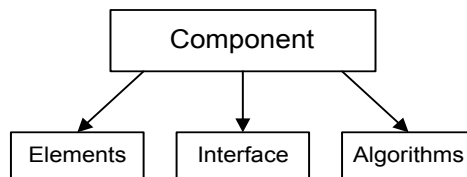


Fig. 2-1 The elements of a component specification

In the following, modules and classes are discussed in general. Then the structure of the interface of a component is explained. The various ways in which algorithms can be described (block diagrams, textual, C code) are discussed in the subsequent chapters. The final sections of this chapter are about a special type of classes: state machines. This special class type can only be described in terms of block diagrams.

2.1 Modules vs. Classes

When specifying an embedded control system, the real-time requirements of the system are crucial. In order to meet these requirements, special components with a real-time capable interface, modules, can be used in ASCET.

A module defines a number of processes; in addition, methods can be defined. A process contains a piece of code, that is executed sequentially. Processes are activated by the operating system, no parameters can be passed. Instead, modules use messages for data exchange, i.e. direct access to a global variable space, which results in a highly efficient communication mechanism.

Unlike processes, which are activated only by the operating system, methods are much more flexible. Each method can have an arbitrary (but fixed) number of arguments and a single return value.

The behavior of modules is unique piece within an embedded control system in the sense that they can be instantiated only once in the context of a project. To avoid this limitation, classes can be used. Classes are object-oriented abstract data types that encapsulate data and make available a well defined interface. The interface is a collection of methods, which can be called from anywhere inside the program. Unlike processes, which can only be activated by the operating system, methods are much more flexible. Each method can have an arbitrary (but fixed) number of arguments and a single return value.

Classes can be instantiated more than once, e.g. more than one accumulator class can exist in a project. Each instance of a class has its own data space (its own parameters and variables), but all instances share the same specification. Global variables defined in classes are the same for all instances of a class (and, in an object-oriented view, can be considered to be class variables), but they can also be accessed by other components.

Classes, however, do not support real-time interprocess communication via messages. This has two reasons. Firstly, classes can have multiple instances and the data consistency scheme of ERCOS^{EK} cannot manage multiple instantiations. Secondly, processes are assigned statically to one fixed task. Whenever a process runs, the operating system creates copies of all its messages. These copies are accessible only to that instance of the process that created them. Hence, if the same message is used by various processes, each process gets its own copy of the message. This strategy is used by the real-time operating system to ensure data consistency over multiple processes.

Methods, on the other hand, can be called arbitrarily from different points in the program, for instance from different processes in different tasks. The method does not "know" the calling task. Thus, it cannot be decided which message copy is relevant for which method call.

The properties of modules and classes are summarized in Tab. 2-1.

Property	Module	Class
Processes	x	
Methods	x	x
Argument passing		x
Messages	x	
Multiple instances		x
Hierarchical design	x	x

Tab. 2-1 The properties of modules and classes

State machines are a special type of class available in ASCET. Their semantic behavior is the same as that of classes, but the notations are different. State machines, for example, have special methods for computing the conditions of a state transition.

When specifying components, modules as well as classes, the structure is often hierarchical, since other previously defined classes or modules are to be reused.

2.2 Definition and Instantiation of Components

A component describes an abstract data type, it makes available an interface, through which it interacts with its environment. When using a component in a project, each element has to be created, i.e. for each element real memory cells have to be allocated. The process of creating an object is also called instantiation. Upon instantiation, the necessary data structure is built and initialized.

Each instance of a component has its own set of elements, but inherits the interface and the functional description from the component itself.

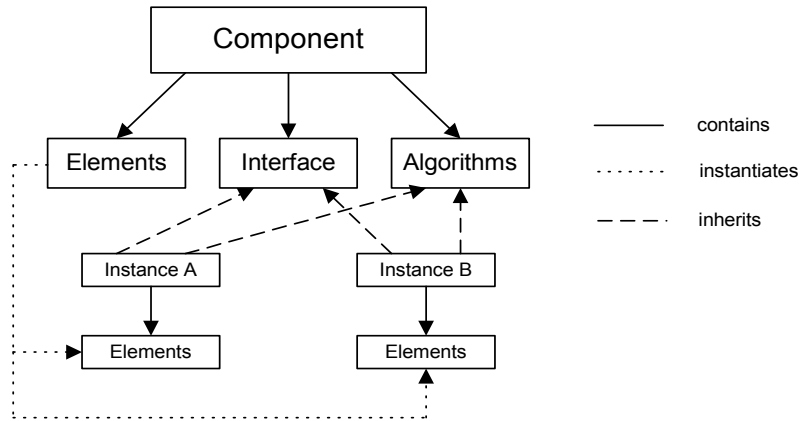


Fig. 2-2 Instantiation and inheritance in components

The definition of a component is therefore the definition of a template for the instantiated components. The difference between template and instance is not obvious for modules, since modules only have one occurrence in a project context, i.e. modules are only instantiated once. There is a one-to-one relation between the template and the instance for modules.

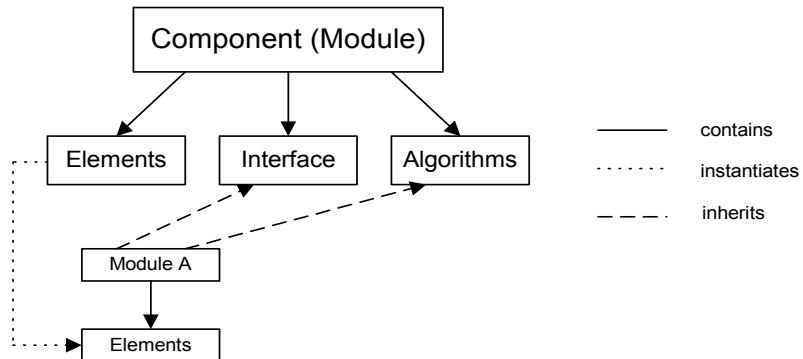


Fig. 2-3 Instantiation and inheritance in modules

Classes, on the other hand, can have multiple instances. Here, the distinction between definition and instantiation becomes more obvious, since there is no simple one-to-one relation between template and instance. The relationship can be 1:n. The definition of a class is therefore the definition of a reusable, user-defined model type.

The instantiation of a component only works in the context of a project. Thus, when working with components only, a default project is automatically created to provide the context for instantiating the components.

When using a class in another component (see following section), the class is instantiated in the context of that component, when that component is instantiated. In contrast to this, modules are always instantiated in a project.

2.3 The Interface of Components

The interface of a component consists of methods, processes, and the access to global variables. Modules, for instance, have access to messages. Methods and processes are structured in the same way. Their structure is independent of the way the methods or processes are described.

Each method or process is assigned to a diagram, where each diagram can either be public or private. Methods assigned to private diagrams are only visible inside the component and do not belong to the public interface of the component, which is visible to other components. All methods assigned to one diagram are described in this diagram (in the case of block diagrams, there is a common block diagram for all these methods).

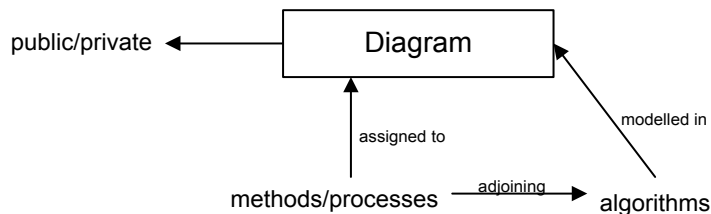


Fig. 2-4 The interface of components

2.3.1 The Interface of Classes

The interface of a class consists of a number of methods which are assigned to one of the diagrams of the class. The interface of each method, consists of its arguments and a return value. Methods are similar to subroutines, that can be called from any point in the software. However, the data encapsulation of a

class, i.e. the access to the same set of instance variables and parameters, makes the concept of methods and classes far more pervasive than that of subroutines. Methods have access to all the elements defined in their class.

The arguments and return value of a method can only be used in the body of the associated method. In addition, each method has a number of method-local variables. These variables are temporary and not static, and like arguments, they can only be used in the body of the associated method.

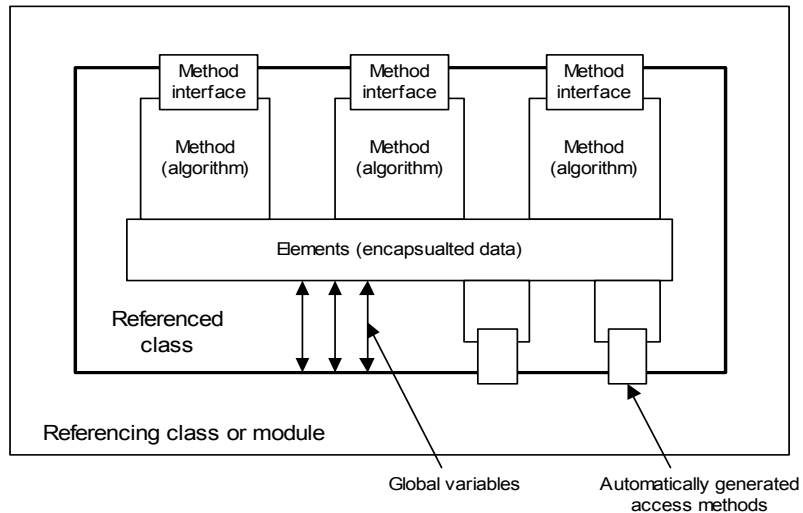


Fig. 2-5 The interface of classes

Additional methods can be made available for direct access to the instance variables of a class. This mechanism allows classes to be used as data containers (similar to records in C).

The interaction of a class with its environment consists of calling the methods of the class. When a method is called, the instructions in the method body are executed.

The methods of a class are categorized as either public or private by assigning them to a public or private diagram. Public methods can be called from any component, that uses that class. Private methods are hidden and can be called only by methods of the same class. They can be used as internal subroutines.

2.3.2 The Interface of Modules

The interface of a module consists of a number of processes and—optional—methods, as well as the messages used in that module. Modules interact at two different levels, since the activation of processes and the communication via messages is separated. The activation of the process is under control of the operating system (that is part of the project).

The communication between processes via messages is asynchronous to the activation of the processes, i.e. the sending of a message and the receiving of it in a process do not happen at the same time. This concept is different from parameter passing between methods, which is synchronous to calling the method.

Like methods, processes can have temporary process-local variables.

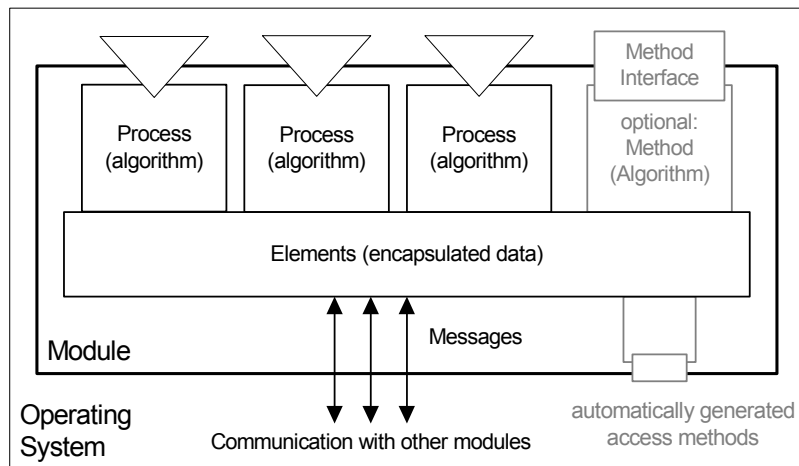


Fig. 2-6 Inter-process communication (grey parts are optional)

2.4 Reusing Components

When specifying components, previously defined classes or modules contain functionality that can be reused. Reusing components leads to a hierarchical, tree-like structure of a component. The leaves of this structure are classes or modules that do not contain other classes or modules.

The structure of a component has to be tree-like, i.e. cyclic dependencies are not allowed. This is because the usage relation is also a containment relation, and a cyclic dependence would be unresolvable.

If a class is used in another component, the class will automatically be instantiated and initialized when the containing component is initialized.

There is, however, an exception. When using a class that is imported, i.e the class is instantiated in some other context, for instance in the project directly, the usage relation is not a containment but a reference relation. Thus a cyclic dependency does not lead to an unresolvable containment relation in this case.

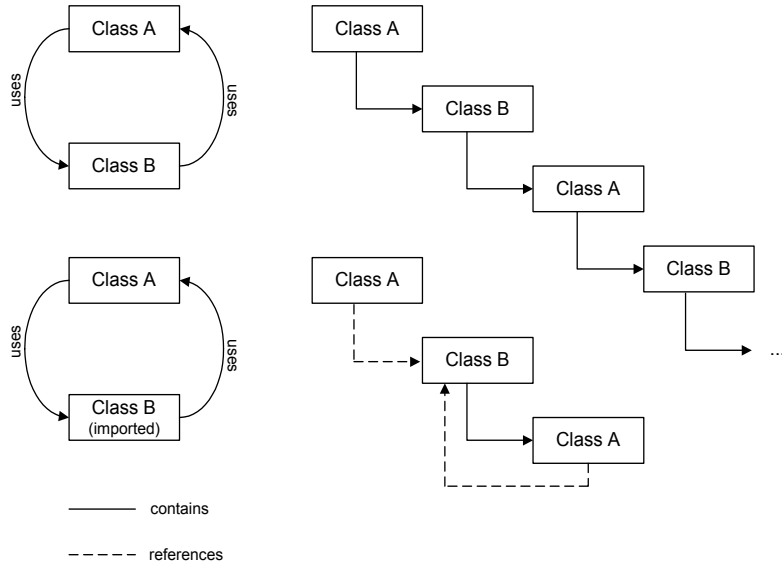


Fig. 2-7 Containment relation and reference

Modules are the top level component. Therefore, modules may not be contained in classes. Classes, however, may be contained in modules as well as in other classes. The following relation holds:

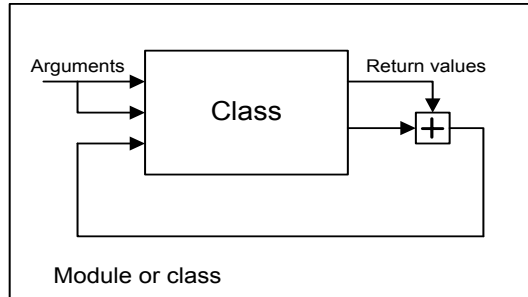
Containment relations	Class	Module
Class	x	-
Module	x	x

Tab. 2-2 Typology of relations

Since the interfaces of modules and classes are different, the meaning of a hierarchical module structure and a hierarchical class structure is also different.

2.4.1 Hierarchical Class Structure

When using a class inside some other component, the methods of the class can be used as subroutines in the component.



Methods are activated by the referencing component

Fig. 2-8 Method invocation in a nested class

The methods are called as part of the execution of the component's methods or process, this point in the software can be determined by the component itself. When calling a method, the component must supply the method with actual parameters for the arguments of the method.

2.4.2 Hierarchical Module Structure

As mentioned before, modules are always instantiated in a project. That is, in a hierarchical module structure, a module used in another module is not instantiated within the containing module. As a consequence, all of the modules instantiated in a project are on the same level, independent of their position in the hierarchical structure.

The hierarchical structuring of modules serves mainly two purposes. A hierarchical structure reflects the nature of a control system. In an engine control, for instance, there may be separate modules for ignition, injection, and lambda control.

In addition, the communication structure in a hierarchical mode can be made much more transparent, since the dataflow is directly visible in block diagrams.

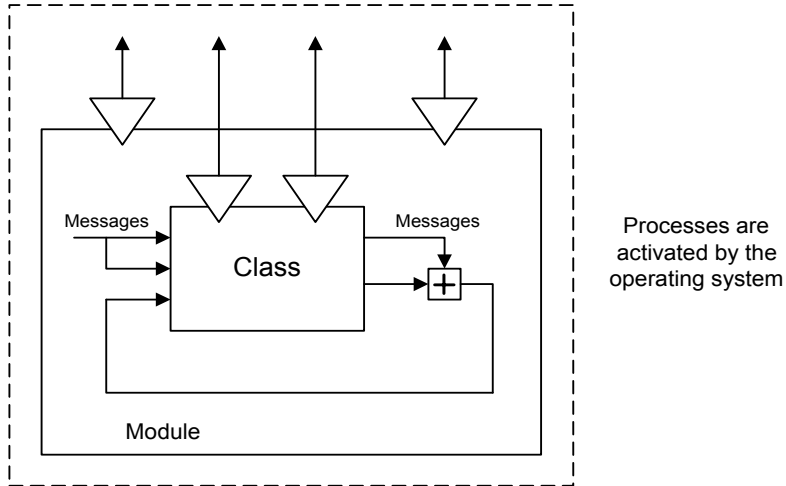


Fig. 2-9 The communication structure in a hierarchical module

A further advantage of a hierarchical module structure becomes clear by this example: easier maintenance. If, for instance, the name of a message is changed, it must be changed in all modules that use that message. If a hierarchical module is used instead, the changes only affect one module, since the name based binding is not explicitly used.

2.5 State Machines

A state machine is a special type of classes, an event-driven system where the focus is not on computations but on control flow. Therefore the main level of description of a state machine, the state diagram, does not describe how data, but how control is passed. To model control flow, a state machine consists of a finite number of states, and transitions between these. Besides, at least one trigger must be included to control the state machine. At each trigger call, one step of the state machine is executed.

For more information on the theory of finite state machines, see

- Harel, David: "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming* 8, 1987, pp. 231-274
- Hatley, Derek J. & Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing Co., Inc., NY, 1988.

The following diagram shows the components of a state machine.

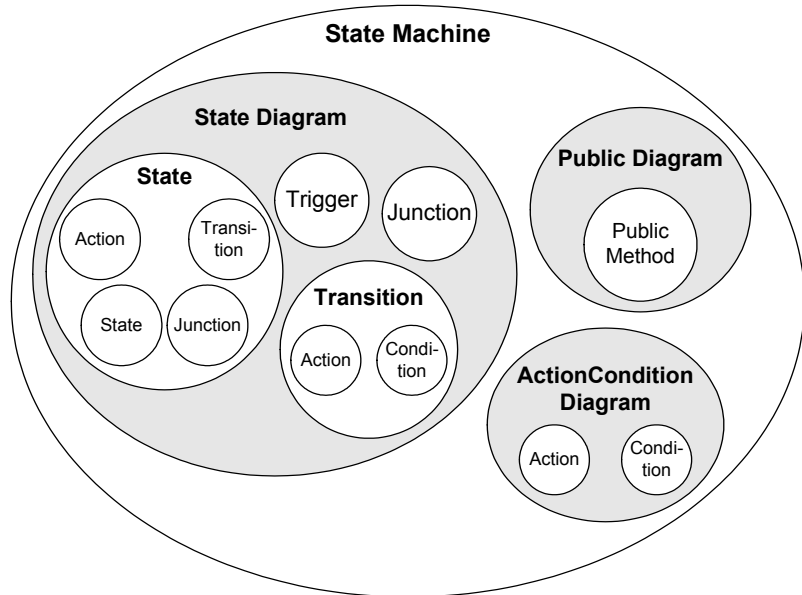


Fig. 2-10 State machine – scheme

Specifying a state machine consists of determining the states a system can be in, defining the conditions that have to be fulfilled for changing from one state to another, and determining the actions that are to be performed during these transitions.

The state diagram is a special block diagram for defining a state machine. Each state is displayed as a rectangle with rounded corners. One of the states always has to be marked as the start state, this is the state the machine is in at the beginning.

The transitions are targeted curves between the states. Each arc represents one transition in a direction marked by an arrowhead at one end. Each end of a transition is connected to a state. The state where the transition starts is the source state, the one where it ends is the destination state. Two arcs are necessary to model a bidirectional transition.

The sample diagram contains the relevant graphical components of a state machine.

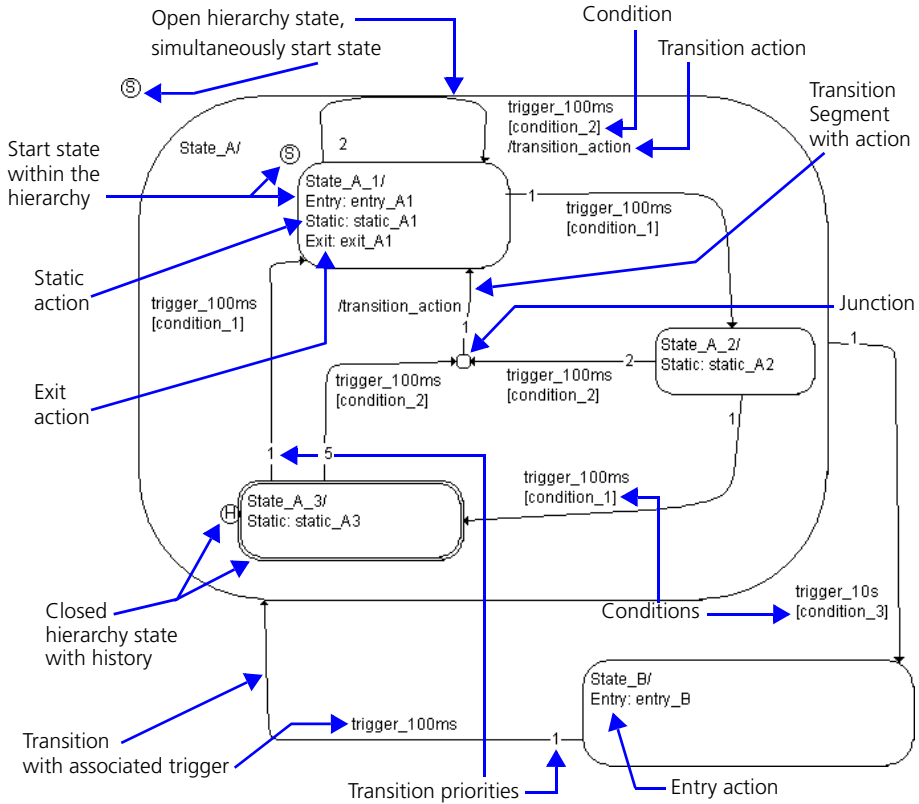


Fig. 2-11 Graphic Components

2.5.1 State Machine Components

States

A *state* describes one mode of an event-driven system. The activity or inactivity of the states changes dynamically, based on trigger events and conditions.

Each state has a parent state (hierarchy state, see page 41). For states on the highest level (State_A and State_B in Fig. 2-11), the state diagram itself is the parent. You can place states within other higher-level states; State_A_1, State_A_2 and State_A_3 are substates of State_A. States containing

no other states are called base states (`State_A_1`, `State_A_2`, `State_B` in Fig. 2-11). A hierarchy state can have a *history* (see page 44). History provides an efficient means of basing future activity on past activity.

The states are mutually exclusive, i.e. only one base state can be active at any one time. If the active base state is the substate of a hierarchy, all hierarchy states that contain the active state are active, too. If, for example, the `State_A_2` state in Fig. 2-11 is active, the hierarchy state `State_A` is active, too. If one of the (invisible) substates of `State_A_3` is active, `State_A_3` and `State_A` are active, too.

Each state has a unique name. Identical names are forbidden within different hierarchies. If you use an existing name a second time, `_n` is added to it. `n` is the smallest unissued number for this name (States `State_A_1` to `State_A_3` in Fig. 2-11). The following names are forbidden, too:

- names of methods, processes, elements etc. in the entire project
- names from the C language (e.g., `static`, `define`, etc.)

Such state names do not always result in an error message, but the generated code is always wrong.

Besides the names, the state labels contain the various *actions* (see page 45). These are processed successively according to their type. The following types exist: entry action, static action and exit action. All actions are optional.

Transitions

A *transition* is a graphic object connecting two states. One end of the transition is attached to the source state where the transition begins. The other is connected to the destination state where the transition ends. A transition may be interrupted by one or more *junctions* (see page 38) and split into several segments.

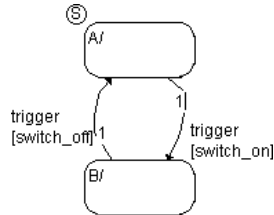
A priority is assigned to each transition. The higher the number, the higher the priority. If more than one transition originate from the same state or junction, they are evaluated in the order of their priorities. Two transitions from the same state may not have the same priority.

A transition label describes the circumstances under which the system moves from one state to another. A trigger event is necessary for a transition to occur. The name of the trigger is the first part of the transition label. In Fig. 2-11, the trigger `trigger_100ms` actuates the transition from `State_A_1` to `State_A_2`. Optionally, the transitions can also contain a *condition* (see page 45) and an action (page 45), the *transition action*. These are named in the second and third part of the label. In the state diagram, conditions are

represented in square brackets, transition actions with a leading "/". How triggers, conditions and actions are assigned to the segments of a transition with junctions is described in "Junctions" on page 38.

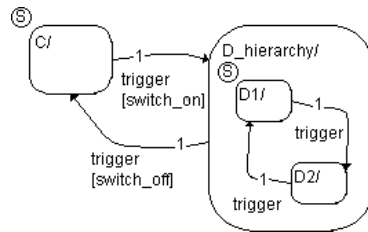
A transition is valid when its source state is active and its condition—if specified—is true. There are several kinds of transitions:

1. Transitions between base states



The transition from state A to B is valid if A is active, the trigger event `trigger` occurs, and the condition `[switch_on]` is true.

2. Transitions from and to hierarchy states

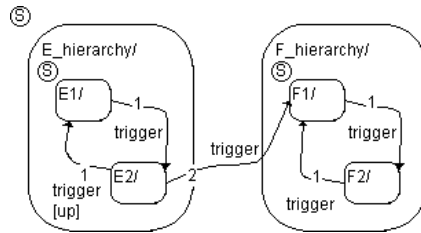


The transition from C to the hierarchy state (see page 41) `D_hierarchy` is valid if C is active, the trigger event `trigger` occurs, and the condition `[switch_on]` is true. It is an explicit transition to the hierarchy state.

For a valid transition to a hierarchy state, you must implicitly define one substate as the destination. Here, you do this by marking the substate D1 as start state (see page 43). What is executed in fact is the transition from C to D1.

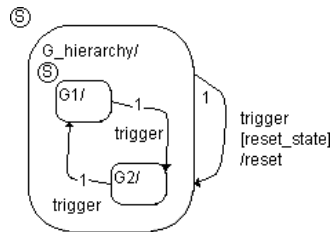
The transition from `D_hierarchy` to C is valid if `D_hierarchy` is active, the trigger event `trigger` occurred, and the condition `[switch_off]` is true, regardless of which substate is active.

3. Transitions between substates of different hierarchies



The transition from the substate $E2$ in the hierarchy state $E_hierarchy$ in the substate $F1$ in the hierarchy state $F_hierarchy$ is valid if $E2$ is active and the trigger event `trigger` occurs. The transition defines an explicit exit from substate $E2$ and an implicit exit from the hierarchy state $E_hierarchy$. It also implicitly defines an entry into $F_hierarchy$ and an entry into $F1$.

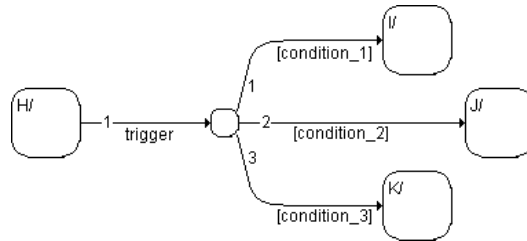
4. Loops



A loop is a transition from a state to itself. The transition in the above figure is valid if either of the substates of $G_hierarchy$ is active, the trigger event `trigger` occurs and the condition `[reset_state]` is true. The system leaves the active substate, it leaves the $G_hierarchy$ state, executes the transition action, re-enters $G_hierarchy$, and finally enters the substate $G1$.

5. Transitions with junctions

All types of transitions can contain junctions (see next section). Here, just one of the many possible examples is shown.



If state H is active and the trigger event `trigger` occurs, the system leaves state H. In the junction, the conditions to the leading transition segments (`[condition_1]`, `[condition_2]`, `[condition_3]`) are tested in sequence for their priority. If, for example, the condition `[condition_2]` is fulfilled, transition to state J occurs. If none of the conditions are fulfilled, the system remains in the start state H.

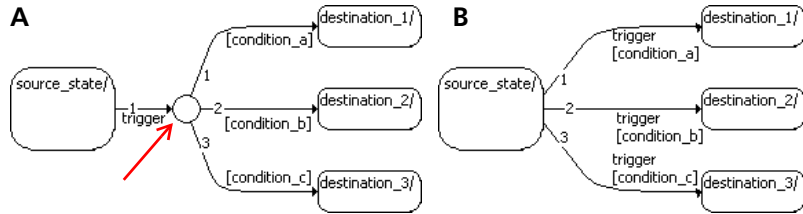
Junctions

A *junction* is a graphic object which considerably improves the legibility of state diagram and aids the generation of efficient code. Junctions form additional possibilities for representing the required system behavior.

Junctions are not states, they represent branching points in the state diagram. Nodes interrupt a transition (see page 35) and split it into segments. One segment connects the source state with the junction, one or more segments connect the interrupting junctions (if required), and the last segment connects the last junction with the destination state. Thus, junctions aid the representation of different transitions. At the same time, they allow reuse of transition segments.

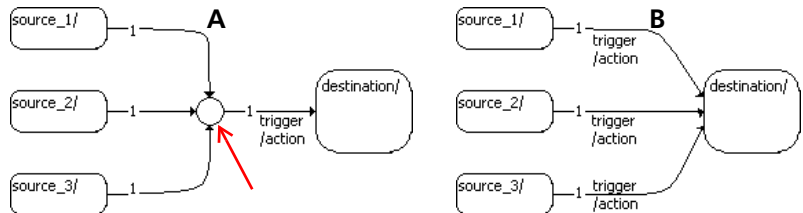
Note the following when using junctions:

- Transitions from a starting state to several destination states are clearly represented.



You can achieve the same functionality modelled with a junction in Part A of the diagram by direct transitions from the start state `source_state` to the destination states (Part B of the diagram). However, using the junction brings a runtime benefit, as the transition segment between the start state and the junction is evaluated first. If this is already invalid, no transition can take place and you need not consider the segments leading away from the junction.

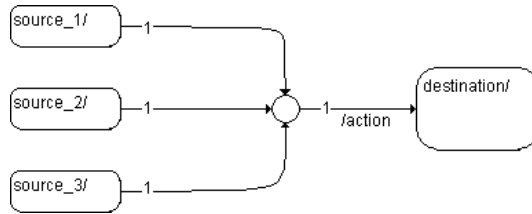
- Also, transitions from several starting states to a destination state are clearly represented.



In this case too, both ways of writing have the same meaning. You can (and should) assign an action shared by all three transitions to the segment leading away from the junction.

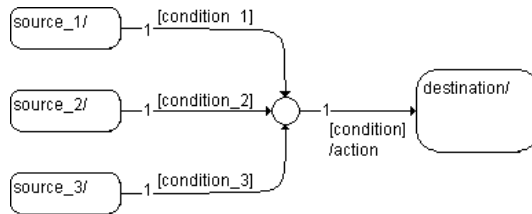
- If none of the transition segments leading away from the junction are valid, then no transition occurs and the system remains in the starting state.

- Transition segments from a junction into a state can contain actions.

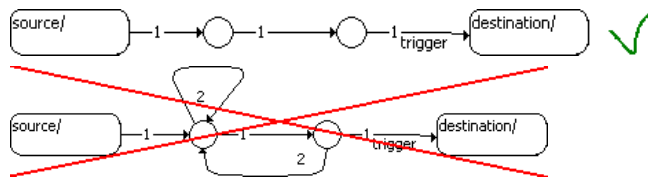


It is not possible to assign an action to a transition segment ending in a junction. The complex semantics of such transition actions results in inefficient coding.

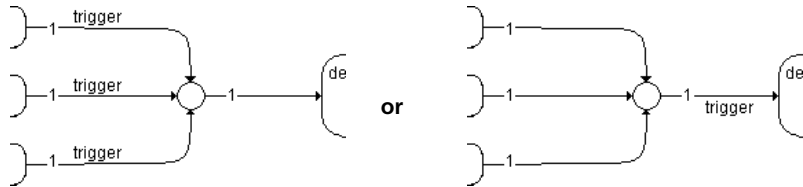
- Each segment of a transition can have a condition.



- Transitions from one junction to another (cascading junctions) are allowed, all kinds of loops are forbidden.



- Only one segment of a transition has a trigger. Usually, a trigger is assigned either to the segments leading towards the first junction of a transition, or to the segments leading away from the last junction, but not to all segments.



Note

The assignment of triggers to more than one segment of the same transition is not deactivated. However, in such a case, ASCET outputs an error message if different triggers are assigned to the segments. You are therefore responsible for the assignment of triggers.

- If none of the segments leading to a possible destination state is valid, no transition occurs. The state remains in the source state.

Triggers

Triggers activate the execution of the state machines: Each trigger call causes the execution of one state machine step. They are public methods of the state machine; you must define each trigger that affects the state diagram. A trigger can have arguments for communication with other ASCET components (see also the sections "State Machines as Classes" on page 83 and "The State Machine Editor" in the ASCET user's guide).

A state machine can have one or more triggers. Each transition is assigned to one of the triggers of the state machine. By this assignment, it is possible to define several substate machines that work on the same states. Each trigger can be started independently. The state machine is activated whenever a trigger is started: all transitions from the current state are checked in the order of their priority, and a transition is executed if necessary.

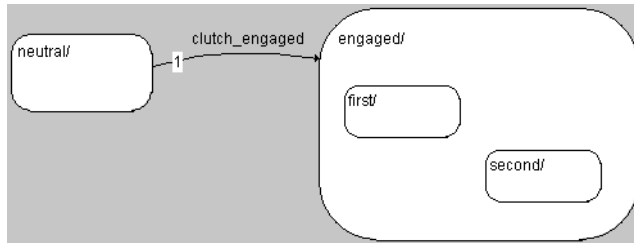
Hierarchy

State machines often have a large number of states. The *hierarchy* allows the organization of complex systems by defining higher or lower-level object structures. A hierarchical design usually reduces the number of transitions and produces structured and readable diagrams (see also section "Hierarchy States" in the ASCET user's guide).

ASCET supports the hierarchical organization of states in the form of open and closed hierarchies (*state_A* and *state_A_3*, respectively, in Fig. 2-11). The only difference between them is the graphical representation.

Each state can contain other states. Those states are called hierarchy states; states containing no other states are called base states. A state contained in a hierarchy state is called a substate of the hierarchy state. The system is always in a base state, and together with that base state also in its associated hierarchy states.

The state diagram shown here has a hierarchy state that contains two substates. (Some transitions are left out for clarity.)



The hierarchy state *engaged* contains the two substates *first* and *second*. This makes *engaged* the parent state of *first* and *second*. When the trigger event *clutch_engaged* occurs, the system transitions from the *neutral* state to the hierarchy state *engaged*.

Far more complicated structures are possible, too (see Fig. 2-11). The following is an example of a hierarchical state machine with two hierarchical substates, one of which contains a further hierarchy state. The lines between the states symbolize a containment relation and should not be confused with transitions.

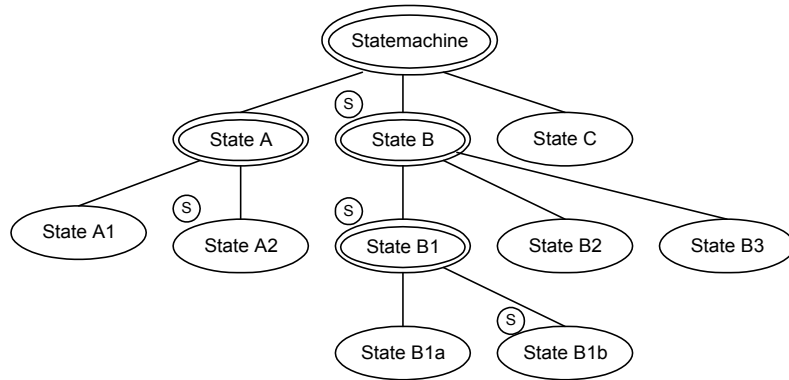


Fig. 2-12 Relationships within complex state machines

Within a hierarchy state, the substates form a state machine of their own. For instance, state A1 and state A2 form a state machine of their own. States inside a hierarchy state can have transitions to other states, which are not located inside the same hierarchy state. The states are connected by transitions; one of the states is marked as the start state in the hierarchy state. At the beginning the hierarchical state machine is in the start state, and if this state is hierarchical, too, it is in the start state of the hierarchical state and so on.

In the above example, the start state of the state machine is state B1b, since it is the start state of state B1, which itself is the start state of B. B, in turn, is the start state on the topmost level.

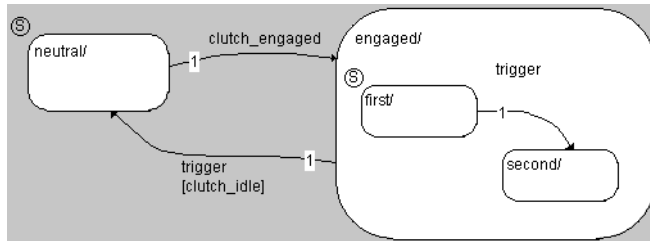
A transition from a hierarchy state automatically includes the exit from the active substate. A transition from a substate can lead beyond the borders of hierarchy states to another substate. If a substate is active, its parent hierarchy state is active, too.

Start State

The *start state* specifies which state is to be activated when there are several possibilities on the same hierarchy level. Thus, the start state of the entire state machine, or that of a hierarchy level is determined.

A common error in the specification of state machines is the generation of several states without marking one of them as start state. In that case, there is no indication of which state becomes active by default. Therefore, on code generation, ASCET outputs an appropriate error message.

The state `neutral` is the start state of the entire state diagram shown below, `first` is the start state of the hierarchy state `engaged`.

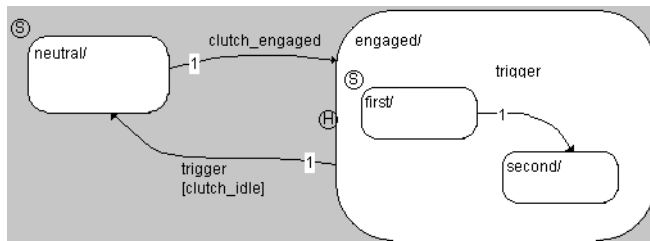


With that, the state `neutral` becomes active when the state machine is first activated. If you had not defined a start state, it would be unclear whether `neutral` or `engaged` should be activated. When a transition from `neutral` to `engaged` occurs, the substate `first` is activated inside the hierarchy state.

History

The *history* option provides the means to determine the destination substate of a transition to a hierarchy state based on past activities. If a hierarchy state has a history, the transition ends in the substate that was most recently active.

The history belongs to the hierarchy state in which the option was set. It takes priority over the start state within the hierarchy.

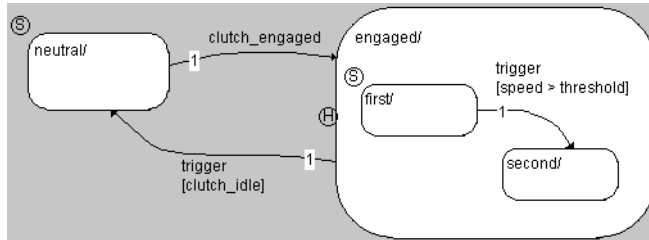


The H in the diagram indicates that the hierarchy state `engaged` has a history. Whether the `first` or `second` substate is activated upon a transition from `neutral` to `engaged` is based on which of them was most recently active.

The generated code contains a special variable for the history, the *history variable*.

Conditions

A *condition* is a Boolean expression specifying that a transition occurs, given that the expression is `true`. Each transition and segment of a transition can have a condition. In Fig. 2-11, the condition `[condition_3]` represents a Boolean expression that must be `true` for the transition from `State_A` to `State_B` to occur.



In the system shown here, the transition from `first` to `second` takes place if the Boolean condition `[speed > threshold]` is true.

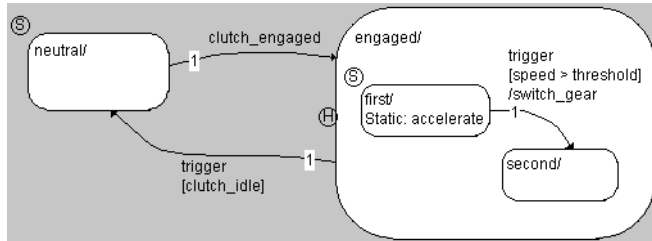
You can specify conditions as block diagrams (in separate diagrams) or in ESDL (in separate diagrams or directly at the transition). For more information, see section "Specifying Conditions and Actions" in the ASCET user's guide.

Conditions can also have arguments for communication with other ASCET components. You can find more on this in section "State Machines as Classes" on page 83 and in the ASCET user's guide, section "Communication with Other Components".

Actions

Actions take place as part of the state machine execution. An action can be executed either as part of a transition from one state to another (e.g. `/transition_action` in Fig. 2-11), or based on the activity status of a state (e.g. `static_A2` or `exit_A1` in Fig. 2-11).

Transitions and transition segments leading away from a junction can have *transition actions*. States can have *entry*, *static* and *exit actions*. All actions are optional. In Fig. 2-11 on page 34, the `State_A_1` state has all three action types, whereas `State_A_2` has neither entry nor exit action, only a static action. The transition from `State_A_1` to `State_A_2` has no action.



When, in this example, the `first` state is active, and no transition occurs, the static action `accelerate` is executed. At the transition from `first` to `second`, the transition action `switch_gear` is executed.

The sections "Semantics: Simple State Machines", "Semantics: Junctions in State Machines" and "Semantics: Hierarchical State Machines" describe in detail which actions are executed when. You can specify actions as block diagrams (in separate diagrams) or in ESDL (in separate diagrams or directly at the transition). For more information, see the ASCET user's guide, section 4.2.3 "Specifying Conditions and Actions".

Actions can also have arguments for communication with other ASCET components. You can find more on this in section "State Machines as Classes" on page 83 and in the ASCET user's guide, section "Communication with Other Components".

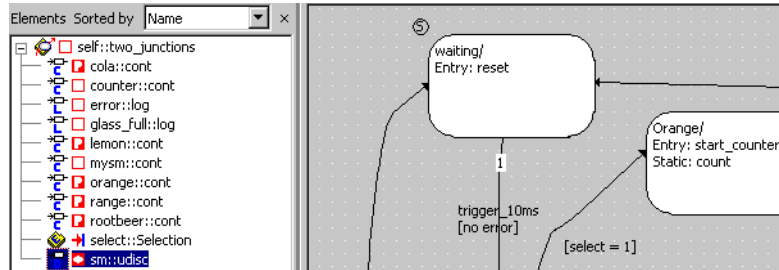
Data

Data objects are used to store and process numerical values in the state diagram. The following types are available:

- Variables, parameters, constants (see page 96)
- Enumerations (see page 96)
- Arrays, matrices (see page 91, 92)
- Literals (see page 96)
- Temporary variables (see page 98)
- Characteristic curves and maps (see page 92)
- Inputs for data from other ASCET components
- Outputs to other ASCET components

- other classes (e.g., timers, counters, comparators)

The state variable `sm` of type `unsigned discrete` also belongs to the data. The variable is created in every state machine.



This variable contains the number of the currently active state. You cannot edit it in the state machine editor but you can measure it in an experiment. If an ASAM-MCD-2CM file is generated for a project containing a state machine, the `sm` parameter is also saved to the file.

2.5.2 Semantics of State Machines

A state machine consists of a finite number of states. Each state represents a state a system can be in, for instance whether a door is locked, open, or closed. Under certain circumstances the state of the system changes. These state changes are modelled by transitions between the different states. For each possible transition to take place, a condition has to be fulfilled.

An external event, the trigger event, activates a state machine. A trigger is a public method of the state machine. A state machine always has to be in one of its states. At the beginning, a state machine is in a special state, the start state. If a trigger event occurs, the system reacts with the execution of actions (e.g., creation of a signal, change of a variable, or transition to another state).

The *entry action* of a state is executed when a transition to that state occurs. The state is activated before the execution of the entry action is started.

Note

*When a state machine is called for the first time, the entry action of the start state is **not** executed.*

The *static action* of a state is executed if the state is active and a trigger event occurs which does not result in a transition from the state. When a transition between two substates of the same hierarchy state occurs, the hierarchy state (which is not left) executes and completes its static action after the source state was left, but before the transition action is executed.

The *exit action* of a state is executed when a transition from that state occurs. The state becomes inactive after the execution of the exit action is completed.

The *transition action* of a transition is executed after the source state has been left and before the destination state is activated.

The semantics describe how a state diagram is interpreted and executed and in which order the actions will be executed. Knowledge of the semantics of state diagram is essential for the creation of suitable state machines and the generation of efficient code. Different implementation options result in different simulation behavior and in the executable code.

The semantics of state machines contain rules for the

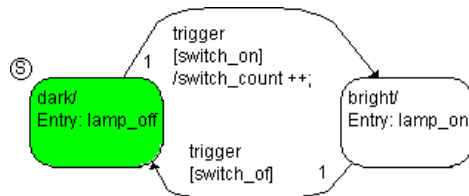
- Processing of states,
- Selection of transitions,
- Processing of transitions.

The following sections describe the semantics of state machines using examples. These cover a wide range of possible implementations and combinations of the different actions.

Refer to the section "Semantics: Summary" on page 68 for a summary of the rules.

2.5.3 Semantics: Simple State Machines

Example 1: Transition between two states.



This simple state machine models a light switch. At the beginning, the lamp is off, the state `dark` is active. The trigger event `trigger` occurs and initiates the evaluation of the state machine. The light switch is pressed, so that the condition `switch_on` is true. The following steps are executed:

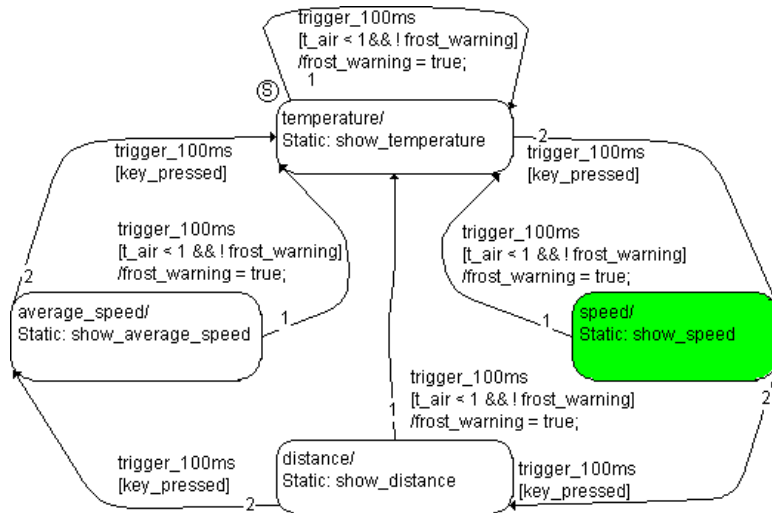
1. The state diagram checks to see if there is a valid transition.
2. The `dark` state is active so that only the transition from `dark` to `bright` has to be evaluated. The condition `[switch_on]` is fulfilled, the transition is valid.
3. The `dark` state has no exit action that could be executed. It is deactivated.

4. The transition action is executed, the counter `switch_count` is increased by 1.
5. The `bright` state is activated.
6. The `lamp_on` entry action is executed and completed. The lamp is switched on.

With that, the evaluation of the state machine initiated by this trigger event is finished.

Every state can have transitions to more than one other state. To make the behavior of the state machine deterministic, each transition has to be assigned a priority. The priority determines the order in which the conditions belonging to the transitions are checked. Once a condition evaluates to `true`, the associated transition takes place, and all other conditions belonging to transitions with lower priorities are not tested. If no condition evaluates to "true", the state remains unchanged and the static action is executed.

Example 2: Several possible transitions from one state

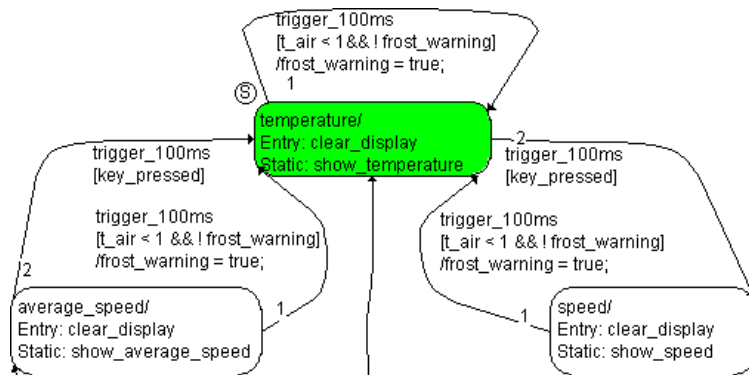


This state machine models a display. Outside temperature, speed, average speed and distance covered can be displayed as required. There is also a key to toggle the display. If the outside temperature falls below 1°C, a change to the temperature display occurs, and a frost warning is shown.

The state machine is in the `speed` state. A trigger event `trigger_100ms` occurs; the temperature drops from 1.5 °C to 0.5 °C. The switch is not pressed. The following steps are executed:

1. The system checks to see if there is a valid transition from `speed`.
2. The transition from `speed` to `distance` has the highest priority, and is evaluated first. However, the `[key_pressed]` condition is not fulfilled, the transition is invalid.
3. The transition from `speed` to `temperature` has the condition `[t_air < 1 && !frost_warning]`. At first, the temperature was above the threshold of 1 °C and no frost warning was required. Now, it has dropped to 0.5 °C. Both parts of the condition are true, the transition is valid.
4. The `speed` state has no exit action. It is deactivated.
5. The `/frost_warning = true` transition action is executed, and the frost warning appears.
6. The `temperature` state is activated.
7. Since that state has no entry action, the evaluation of the state machine initiated by this trigger event is finished.

Example 3: Loop



The state machine is the same as in Example 2. However, the entry action `clear_display` was added to the states. The state machine is in the `temperature` state. Otherwise, the starting state is the same as in the previous example. A trigger event `trigger_100ms` occurs and the switch is not pressed. The following steps are executed:

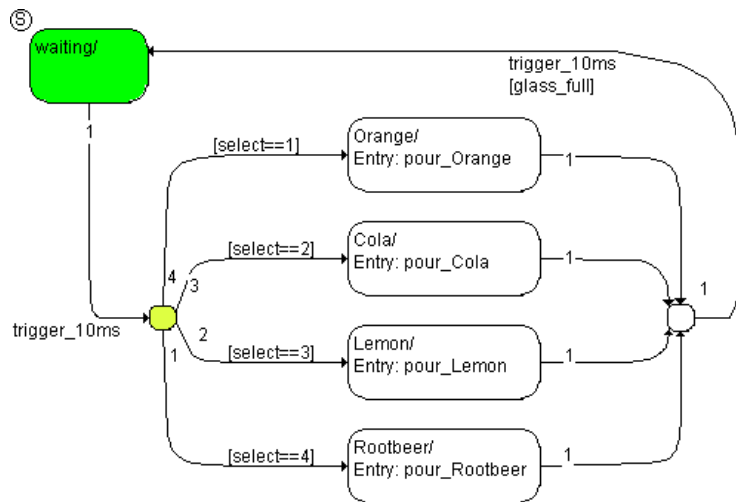
1. The system checks to see if there is a valid transition from `temperature`.
2. The transition from `temperature` to `speed` has a higher priority, but the condition is not fulfilled. The transition is invalid.
3. The transition from `temperature` to itself has the condition `[t_air < 1 && !frost_warning]`. This is fulfilled, the transition is valid.
4. The `temperature` state has no exit action. It is deactivated.
5. The `/frost_warning = true` transition action is executed, and the frost warning appears.
6. The `temperature` state is activated.
7. The entry action `clear_display` of the `temperature` substate is executed and completed.

With that, the evaluation of the state machine initiated by the `trigger_100ms` trigger event is finished.

2.5.4 Semantics: Junctions in State Machines

Junctions (see page 38) aid the legibility of state diagrams. The functionality of all the examples can also be described using direct transitions between the states.

Example 4: If...Then...Else Construction



This state machine models a simple drinks machine which offers four different drinks. The state machine is in the `waiting` state. A trigger event `trigger_10ms` occurs: someone wants Cola. This sets the `select` selection to 2. The following steps are executed:

1. The system checks to see if there is a valid transition or a valid segment from `waiting`.
The transition segment from `waiting` to the left-hand junction is valid.
2. The transition segments leading away from the junction are examined in order of their priority, starting with the segment of the junction to state `Orange`.
The condition `[select==1]` is not fulfilled, the segment is invalid.
3. Next, the segment from the junction to state `Cola` is tested.
The condition `[select==2]` is fulfilled, the segment is valid. This means that there is a fully-valid transition available from the state `waiting`.
4. Only now does the transition occur. The state `waiting` has no exit action and is deactivated.
5. The `Cola` state is activated.
6. The `pour_Cola` entry action is executed and completed.
With that, the evaluation of the state machine initiated by this trigger event is finished.

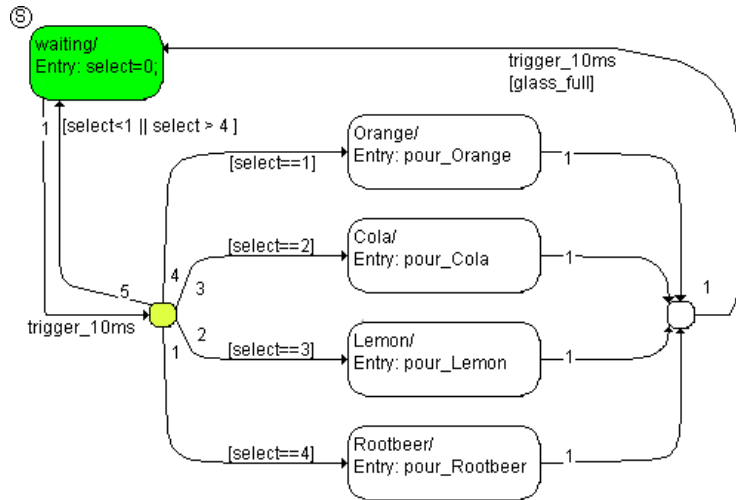
Example 5: No transition

The state machine is the same as in Example 4. The state machine is in the `speed` state. A trigger event `trigger_10ms` occurs, the selection `select` is set to 5 by mistake. The following steps are executed:

1. The system checks to see if there is a valid transition or a valid segment from `waiting`.
The transition segment from `waiting` to the left-hand junction is valid.
2. The transition segments leading away from the junction are examined in the order of their priority.
As `select` was set to 5, none of the conditions are fulfilled, all the segments are invalid.

- There is no valid transition from `waiting`. The system remains in the state `waiting`. As the state has no static action, nothing happens. With that, the evaluation of the state machine initiated by this trigger event is finished.

Example 6: Loop construction



The state machine is the same as in Example 4. The addition is a transition segment away from the junction back to the state `waiting` and the entry action in `waiting`.

The state machine is in the state `waiting`; a trigger event `trigger_10ms` occurs. By mistake the selection `select` is set to 5. The following steps are executed:

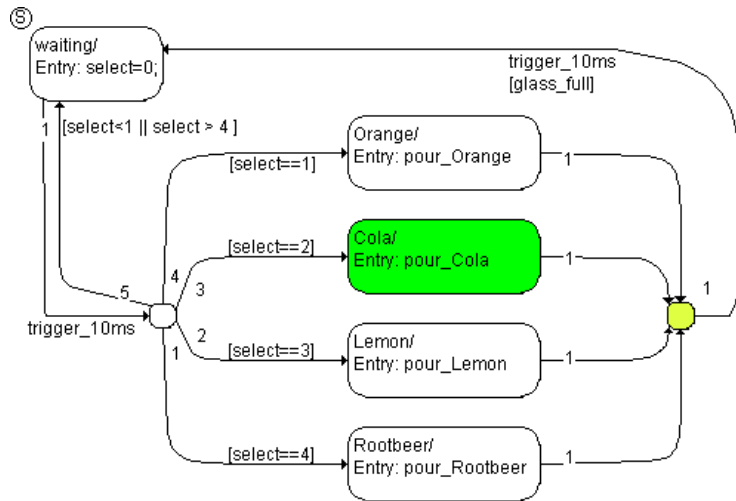
- The system checks to see if there is a valid transition or a valid segment from `waiting`.
The transition segment from `waiting` to the left-hand junction is valid.
- The transition segments leading away from the junction are examined in the order of their priorities, starting with the segment of the junction back to the state `waiting`.
The condition `[select<1 || select > 4]` is fulfilled, the segment is valid. This means that there is a complete, valid transition available from the state `waiting`.

3. The `waiting` state has no exit action. It is deactivated.
4. The transition from `waiting` to `waiting` has no transition action, and therefore the state `waiting` is reactivated.
5. The entry action `select=0;` from `waiting` is executed and completed.

With that, the evaluation of the state machine initiated by this trigger event is finished.

This loop construction corresponds to a direct transition from a state to itself from Example 3.

Example 7: Transitions from multiple start states to a destination state (one trigger)



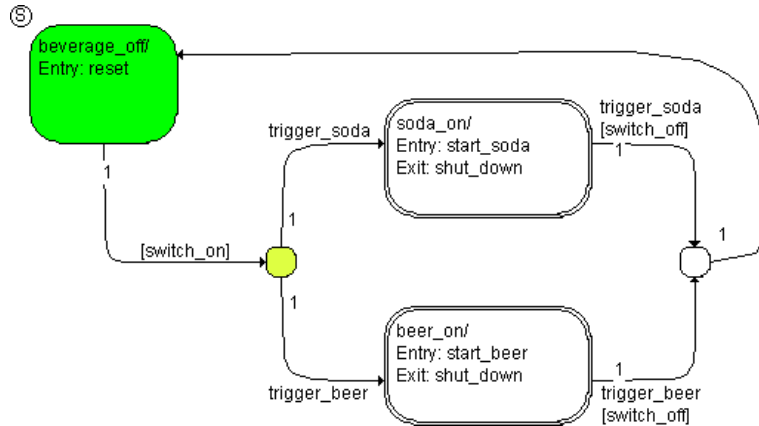
The state machine is the same as in Example 6. The state `Cola` is active, the glass has been filled and the logical variable `glass_full` set to `true`. A trigger event `trigger_10ms` occurs, and the following steps are performed:

1. The system checks to see if there is a valid transition or a valid segment from `Cola` available.
The transition segment from `Cola` to the right-hand junction is valid.
2. The transition segment from the junction to the state `waiting` has the condition `[glass_full]`. As `glass_full` was set to `true`, this segment is also valid and the transition can take place.
3. The `Cola` state has no exit action. It is deactivated.

4. The transition has no transition action and therefore the state `waiting` is activated next.
5. The entry action `select=0` from `waiting` is executed and completed.

With that, the evaluation of the state machine initiated by this trigger event is finished.

Example 8: Transitions from a start state to different destination states (multiple triggers)



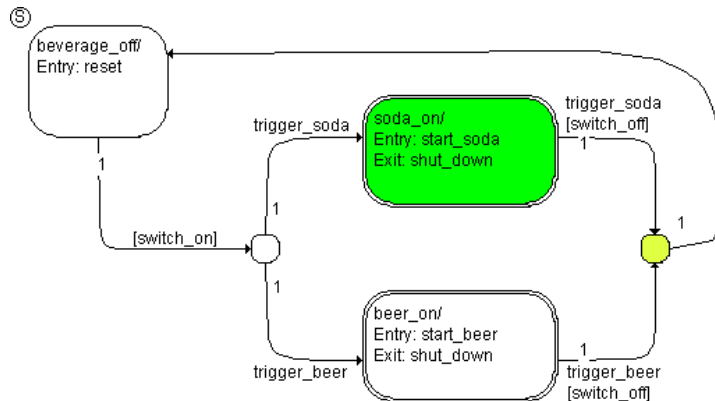
This state machine describes a drinks machine which offers different types of sodas or beers. The actual choice takes place in the hierarchy states `soda_on` and `beer_on`; it is irrelevant for the example. The section "Semantics: Hierarchical State Machines" describes the semantics of hierarchical state machines.

The state machine is in the starting state `beverage_off`. A trigger event `trigger_soda` occurs and the machine is switched on (`switch_on` is true). The following steps are executed:

1. The system checks to see if there is a valid transition or a segment from `beverage_off`.
2. The transition segment from `beverage_off` to the junctions is valid, as the condition `[switch_on]` is fulfilled. As the trigger event `trigger_soda` has occurred, the segment from the junction in the state `soda_on` is also valid; the transition can occur.
3. The `beverage_off` state has no exit action. It is deactivated.
4. The transition from `beverage_off` to `soda_on` has no transition action. Therefore, the state `soda_on` is activated next.

5. The entry action `start_soda` of `soda_on` is executed and completed.
 6. The necessary steps in the hierarchy state are executed.
- With that, the evaluation of the state machine initiated by this trigger event is finished.

Example 9: Transitions from different start states to the same destination state (multiple triggers)



The state machine is the same as in Example 8. The system is in the state `soda_on` (or in one of the substates of the hierarchy). A trigger event `trigger_soda` occurs, the machine is switched off (`switch_off` is true). The following steps are executed:

1. The system checks to see if there is a valid transition or a segment from `soda_on` available.
2. The transition segment from `soda_on` to the junctions is valid, as the condition `[switch_off]` is fulfilled. As the trigger event `trigger_soda` has occurred, the segment from the junction in the state `beverage_off` is also valid; the transition can occur.
3. The necessary steps in the hierarchy state are executed.
4. The exit action `shut_down` of the state `soda_on` is executed.
5. The transition from `soda_on` to `beverage_off` has no transition action. Therefore, the state `beverage_off` is activated next.

6. The entry action `reset` of `beverage_off` is executed and completed.

With that, the evaluation of the state machine initiated by this trigger event is finished.

2.5.5 Semantics: Hierarchical State Machines

Upon activation of the state machine, the conditions of the transitions are checked. The hierarchical order determines the priority. The highest hierarchical level has the highest priority, i.e. the conditions on transitions on upper hierarchy levels are checked first. When a hierarchy state is left, the current substates are left as well. The innermost substate is left first, the outermost hierarchy state is left last. When entering a hierarchy state, the order in which the entry actions are executed is from the outermost hierarchy state to the innermost (base) state, i.e. the outermost state is entered first, and the innermost is entered last. If no transition takes place, the static actions are executed in an outward sequence, i.e. the static action of the innermost substate is executed first, and the static action of the outermost hierarchy state is executed last.

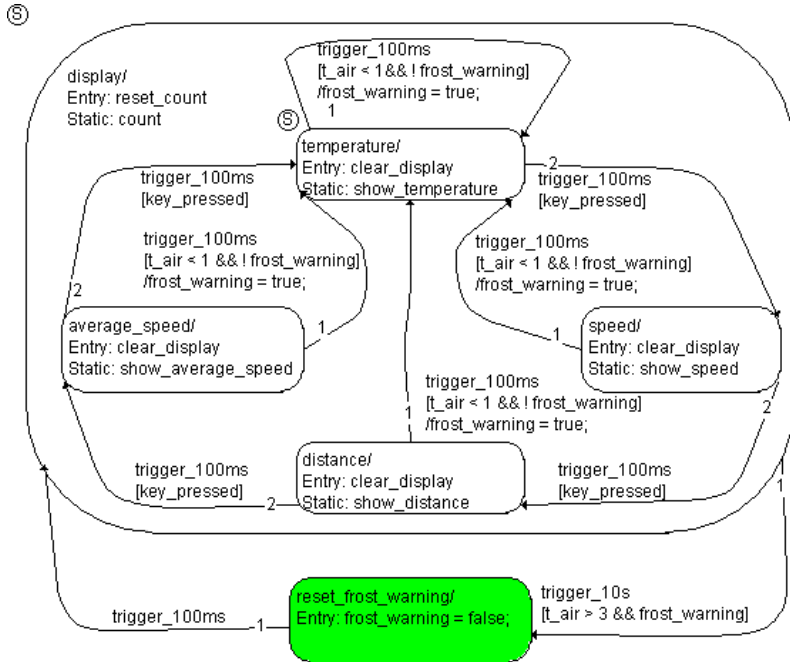
Note

*The examples in this chapter assume **no** optimization of static actions in hierarchy states. If this optimization is activated, the semantics change, see "Optimized for Code Size" on page 76*

Example 10: Transition to a hierarchy state without history

On entry into a hierarchical level, there are two possibilities: either entry into the start state of the hierarchy state (this example). In this case, the hierarchy state has 'forgotten' the substate it has been in when it was left. Alternatively, the last active substate is entered. In this case, the hierarchy state has a history (Example 11).

For each hierarchy state it is possible to determine whether it has a history or not. When entering a hierarchy state with history for the first time, the start state of that hierarchy state is entered.



In the state `display`, this hierarchical state machine contains the display function from Example 3. `display` is a hierarchy state. As soon as a temperature of 3 °C is exceeded, the frost warning is to be reset. The second state on the highest hierarchy level, `reset_frost_warning`, is used for that purpose. Every 10 seconds, a change from `display` to the `reset_frost_warning` state can occur, where the frost warning is switched off.

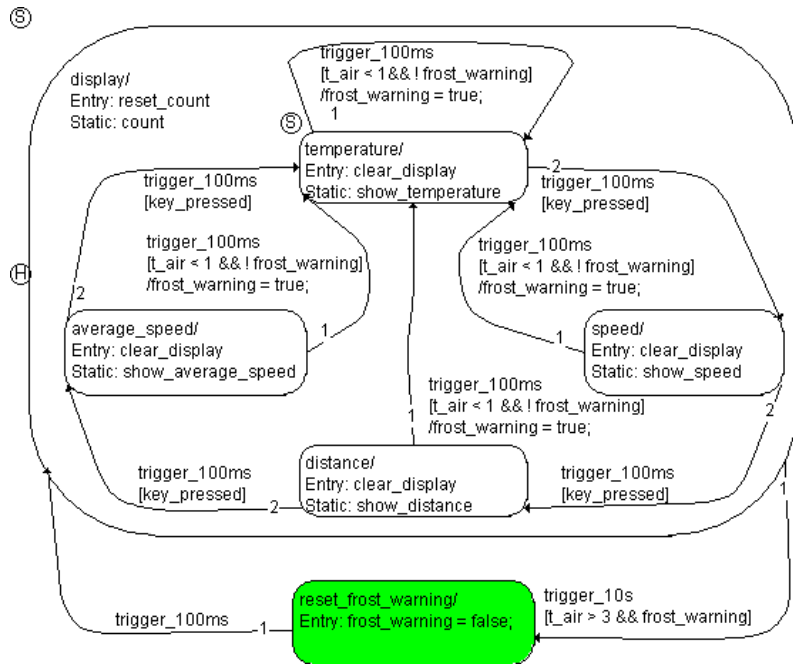
After the frost warning was displayed (`frost_warning = true`), the distance display was selected so that the system was in the distance state. The temperature rose to 5 °C, and the transition from `display` to `reset_frost_warning` took place when the trigger event `trigger_10s` occurred. The system is now in the `reset_frost_warning` state. A trigger event `trigger_100ms` occurs, and the following steps are performed:

1. The system checks to see if there is a valid transition from `reset_frost_warning`.
2. The transition from `reset_frost_warning` to `display` has no condition; it is therefore valid at every `trigger_100ms` trigger event.
3. The `reset_frost_warning` state has no exit action. It is deactivated.

4. The transition from `reset_frost_warning` to `display` has no transition action, and the `display` hierarchy state is activated next.
5. The `reset_count` entry action of the `display` hierarchy state is executed and completed.
6. The `temperature` substate is the start state in the hierarchy. It is activated.
7. The entry action `clear_display` of the `temperature` substate is executed and completed.

With that, the evaluation of the state machine initiated by the `trigger_100ms` trigger event is finished.

Example 11: Transition to a hierarchy state with history



The state machine is the same as in Example 10. Now it has a history. The prehistory and the starting state are the same as in the previous example.

The system is now in the `reset_frost_warning` state. A trigger event `trigger_100ms` occurs, and the following steps are performed:

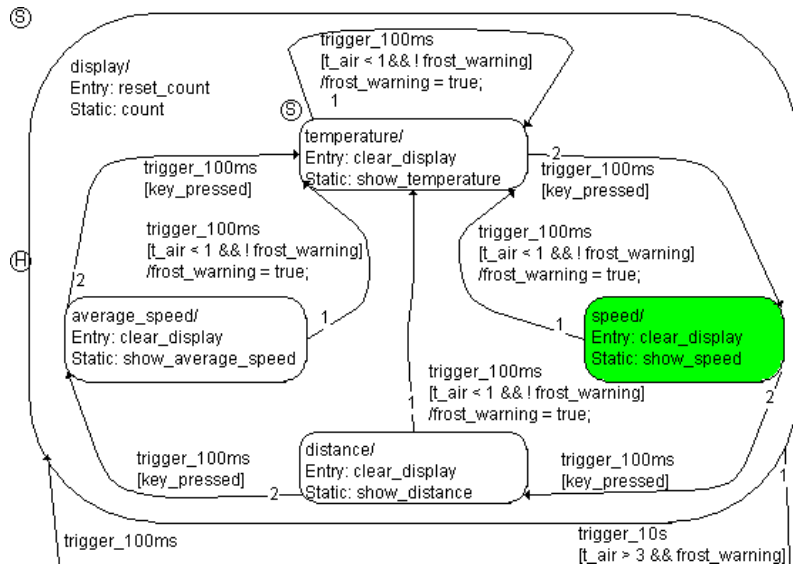
1. The system checks to see if there is a valid transition from `reset_frost_warning`.

2. The transition from `reset_frost_warning` to `display` has no condition; it is therefore valid at every `trigger_100ms` trigger event.
3. The `reset_frost_warning` state has no exit action. It is deactivated.
4. The transition from `reset_frost_warning` to `display` has no transition action, and the `display` hierarchy state is activated next.
5. The `reset_count` entry action of the `display` hierarchy state is executed and completed.
6. Since `display` has a history ('H' in the above figure), the `speed` substate is activated. That state was active when the hierarchy state was left.
7. The entry action `clear_display` of the `speed` substate is executed and completed.

With that, the evaluation of the state machine initiated by the `trigger_100ms` trigger event is finished.

Example 12: Transition within a hierarchy state

If a transition takes place inside a hierarchy state, the state machine remains in that hierarchy state. Therefore, the static action of the hierarchy state is executed, as well as the static actions of all hierarchy states that contain the state in question. They are executed after all exit actions, and before the transition action, from the innermost hierarchy state to the outermost one.



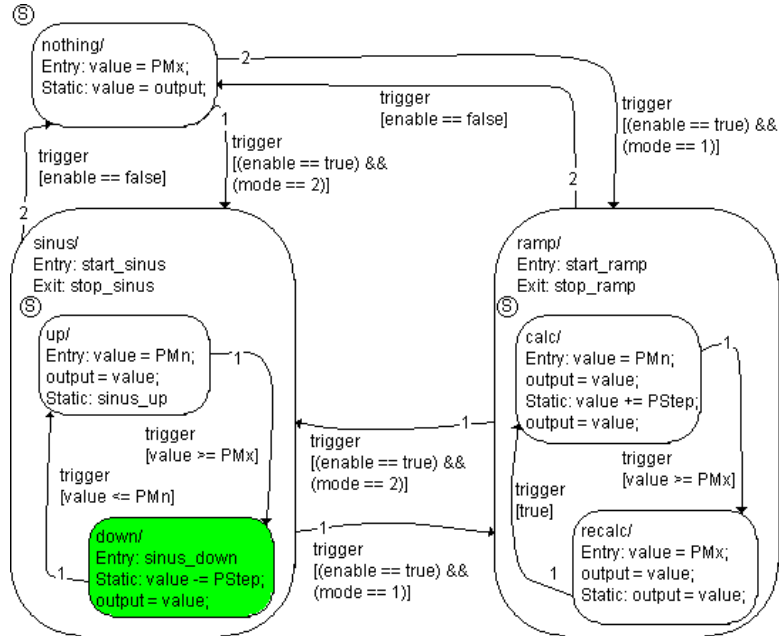
The state machine is the same as in Example 11. The state machine is in the `speed` state. The temperature is still 5 °C, the frost warning is switched off (`frost_warning` is `false`). A trigger event `trigger_100ms` occurs, the switch is pressed (`key_pressed` is `true`). The following steps are executed:

1. The system checks to see if there is a valid transition.
2. The transition from the `display` hierarchy state to `reset_frost_warning` is initiated by another trigger (`trigger_10s`); it is of no importance here.
3. The transition from `speed` to the `distance` substate is evaluated. The condition `[key_pressed]` is fulfilled, the transition is valid.
4. The `speed` state has no exit action. It is deactivated.
5. The `display` hierarchy state is not left. Therefore, its static action `count` is executed and completed.

6. The transition from `speed` to `distance` has no transition action, and the `distance` substate is activated.
7. The entry action `clear_display` of the `speed` substate is executed and completed.

With that, the evaluation of the state machine initiated by the `trigger_100ms` trigger event is finished.

Example 13: Transition between hierarchy states



This state machine acts as a data generator. When `enable` is set to `true`, a signal is produced, either a ramp (state `ramp`, `mode = 1`) or a sine (state `sinus`, `mode = 2`).

The `down` substate in the `sinus` hierarchy state is active. The signal `mode` is set to 1, `enable` remains `true`. A trigger event occurs, and the following steps are performed:

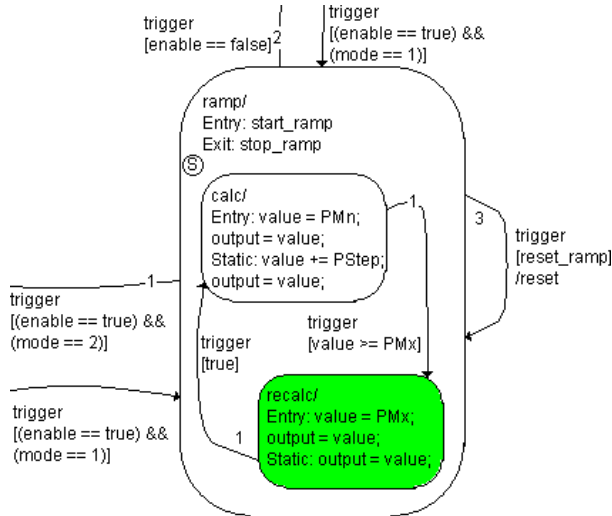
1. The system checks to see if there is a valid transition. Since the transitions from the `sinus` hierarchy state have higher priorities than those from `down`, they are evaluated first.

2. The transition from `sinus` to `nothing` has the highest priority. It is invalid, though, because the condition `[enable == false]` is not fulfilled.
3. The transition from `sinus` to `ramp` is evaluated next. The condition `[(enable == true) && (mode == 1)]` is true, the transition takes place.
The transition from the `down` substate to the `up` substate has the lowest priority and is not evaluated.
4. The `down` substate has no exit action, it is deactivated immediately.
5. The exit action `stop_sinus` of the `sinus` hierarchy state is executed and completed.
6. The `sinus` hierarchy state is deactivated.
7. The transition from `sinus` to `ramp` has no transition action, therefore the `ramp` hierarchy state is activated next.
8. The entry action `start_ramp` of `ramp` is executed and completed.
9. The `calc` substate is the start state within the hierarchy. It is activated.
10. The entry action `value = PMn; output = value;` of `calc` is executed and completed.

With that, the evaluation of the state machine initiated by this trigger event is finished.

Example 14: Loop

The source and destination states of a transition can be identical. Such loops are frequently used to specify the reset function of a hierarchy state.



The `ramp` hierarchy state from the state machine in Example 13 has now a reset function in the form of a loop, i.e. a transition from `ramp` to itself. The rest of the state diagram is left out for clarity.

The `recalc` substate in the `ramp` hierarchy state is active. A trigger event occurs, the reset button is pressed (`reset_ramp = true`). `enable` and `mode` remain unchanged. The following steps are executed:

1. The system checks to see if there is a valid transition.
2. The loop has the highest priority. The condition `[reset_ramp]` is fulfilled, the transition is valid.

Other transitions are not evaluated.

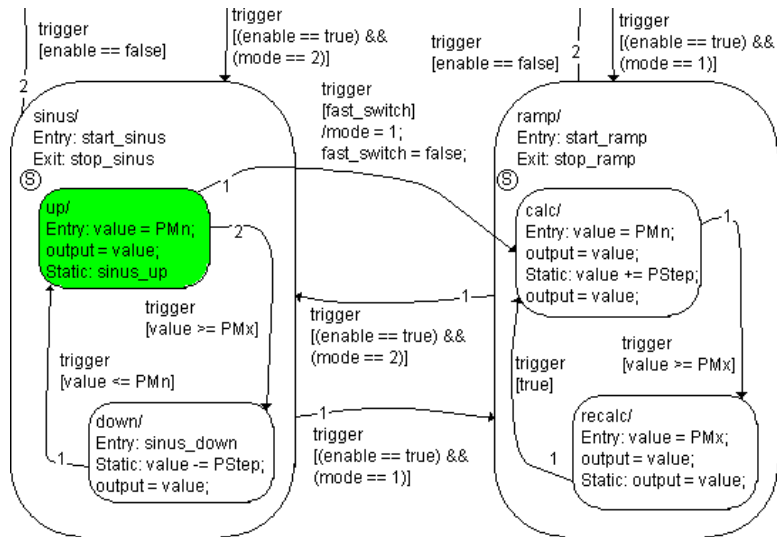
3. The `recalc` substate has no exit action, it is deactivated immediately.
4. The exit action `stop_ramp` of the `ramp` hierarchy state is executed and completed.
5. The `ramp` hierarchy state is deactivated.
6. The loop's transition action `/reset` is executed and completed.
7. The `ramp` hierarchy state is re-activated.
8. The entry action `start_ramp` of `ramp` is executed and completed.

9. The `calc` substate is the start state within the hierarchy. It is activated.
10. The entry action of `calc` is executed and completed.

With that, the evaluation of the state machine initiated by this trigger event is finished.

Example 15: Transition between substates of different hierarchies

Transitions can lead directly from the substate of one hierarchy state to the substate of another hierarchy state.



This state machine is the same as the one in Example 13, only the transition from the `up` substate in the `sinus` to the substate `calc` in `ramp` was added.

The `up` substate in the `sinus` hierarchy state is active. The value `value` is lower than the maximum `PMx`. A trigger event occurs. `mode` remains `2`, and `enable` remains `true`, but the fast-switch is pressed (`fast_switch = true`). The following steps are executed:

1. The system checks to see if there is a valid transition.
2. The transitions from `sinus` to `nothing` and from `sinus` to `ramp` are evaluated first. They are both invalid because the associated conditions are not fulfilled.
3. The transition from substate `up` to substate `down` is evaluated next. It is invalid, too, because the condition `[value >= PMx]` is not fulfilled.

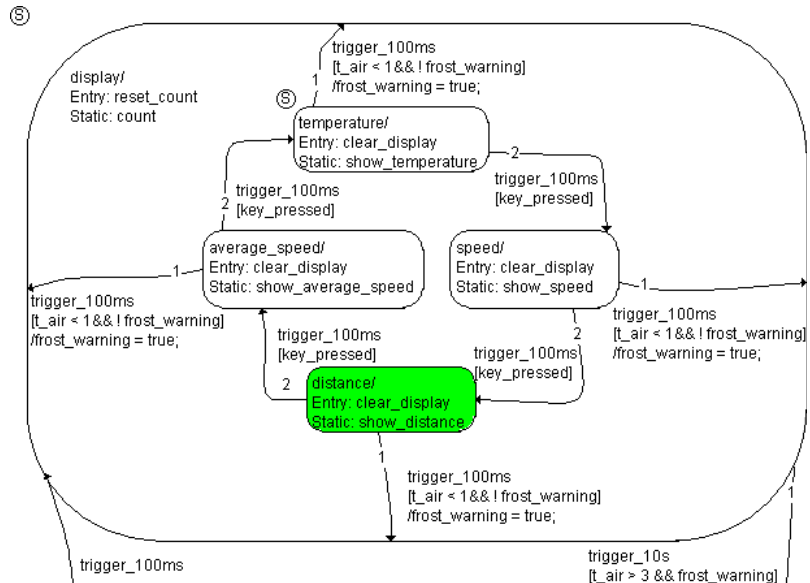
4. The transition from `up` to the `calc` substate has the lowest priority and is evaluated last. The condition `[fast_switch]` is true, the transition takes place.
5. The `up` substate has no exit action; it is deactivated immediately.
6. The exit action `stop_sinus` of the `sinus` hierarchy state is executed and completed.
7. The `sinus` hierarchy state is deactivated.
8. The transition action `(/mode = 1; fast_switch = false;)` is executed and completed.
9. The `ramp` hierarchy state is activated.
10. The entry action of `ramp` is executed and completed.
11. The `calc` substate is activated.
12. The entry action of `calc` is executed and completed.

With that, the evaluation of the state machine initiated by this trigger event is finished.

Example 16: Transition from a substate to a hierarchy state

If the transition from a substate does not lead to another substate, but to the hierarchy state, the procedure is almost the same. The substate is left, the hierarchy state is left, too, and immediately re-entered. Depending on whether the

hierarchy state has a history, either the most recently activated substate or the start state of the hierarchy is entered. This is another way to realize, for example, the frost warning.



The state machine is very similar to the Example 10 only here, the frost warning is implemented using transitions in the hierarchy state. It is in the state `distance`. `frost_warning` is false. A trigger event `trigger_100ms` occurs; the temperature drops to 0.5 °C. The switch is not pressed. The following steps are executed:

1. The system checks to see if there is a valid transition from `distance`.
2. Another trigger initiates the transition from `display` to `reset_frost_warning`; it is of no importance here.
3. The transition from `distance` to `average_speed` is evaluated. The condition `[key_pressed]` is not fulfilled, the transition is invalid.
4. The transition from `distance` to `display` has the condition `[t_air < 1 && !frost_warning]`. Both parts of the condition are true, the transition is valid.
5. The `distance` state has no exit action. It is deactivated.
6. The `display` hierarchy state has no exit action. It is deactivated.
7. The `display` hierarchy state is activated again.

8. The `reset_count` entry action of `display` is executed and completed.
 9. The `/frost_warning = true` transition action is executed, and the frost warning appears.
 10. The `temperature` state is the start state in the hierarchy. It is activated as `display` does not have a history.
 11. The entry action `clear_display` of the `temperature` substate is executed and completed.
- With that, the evaluation of the state machine initiated by the `trigger_100ms` trigger event is finished.

Example 17: No transition

The state machine is the same as in Example 16. The state machine is in the `temperature` state. The temperature is unchanged. A trigger event `trigger_100ms` occurs, the switch is not pressed (`key_pressed` is `false`). The following steps are executed:

1. The system checks to see if there is a valid transition.
2. Another trigger initiates the transition from `display` to `reset_frost_warning`; it is of no importance here.
3. The transition from `temperature` to `speed` is invalid because the switch was not pressed.
4. The transition from `temperature` to `display` is invalid because `frost_warning = true` and thus the condition is false.
There are no other possible transitions available.
5. The static action `show_temperature` in the `temperature` substate is executed and completed.
6. The static action `count` in the hierarchy state `display` is executed and completed.

With that, the evaluation of the state machine initiated by the `trigger_100ms` trigger event is finished.

2.5.6 Semantics: Summary

Initialization of the state diagram: The start state of the system is activated. If the start state is a hierarchy state, the start state within the hierarchy is also activated. No entry action is executed.

Entering a state:

1. If the state has an inactive higher-level state, steps 1–4 are executed for that state.

2. The state is activated.
3. The entry action is executed.
4. Carry out implicit entry actions as necessary:
 - 4.1 If the state contains a subordinate diagram with a history, and if one of the substate was active after initialization, this substate is activated and its entry action executed.
 - 4.2 If the state contains a subordinate diagram with a history, and if one of the substate was active after initialization, this substate is activated and its entry action executed. Otherwise, proceed as described in 4.1.

Executing a (basis) state:

1. The transitions leading away from the state and transitions leading out of higher-level states are evaluated in order of their priority.
2. If a valid transition is found, it is executed. This ends the execution of the state.
3. If no valid transition from the state is available, the static action is executed.
4. If the state has higher-level states, their static actions are executed.

Leaving a state:

1. If the state contains active substates, their exit actions are executed. The exit action of the innermost basis state is executed first.
2. The exit action of the state is executed.
3. The state is deactivated.

Executing a transition:

The transitions are evaluated in the order of their priority. Transitions from a hierarchy state always have a higher priority than transitions from the sub-states of this hierarchy state.

1. A transition or transition segment is tested.
2. If the transition/segment is invalid, the transition/segment with the next-lowest priority is tested.
3. If the transition/segment is valid, the next step depends on where the transition/segment ends.

In a state:

- 3.1 No additional transitions or transition segments are tested. In the case of a transition segment from a junction, the segment is pulled in to the junction in question to obtain a complete transition.
- 3.2 The substates of the start state are left (see "Leaving a state").
- 3.3 The start state is left.
- 3.4 The transition action is executed.
- 3.5 The system enters the destination state (see "Entering a state").

In a junction:

- 3.1 The transition segments leading away from the junction are evaluated as described in steps 1 – 3.
4. If all the transition segments leading away from a junction are invalid, the system returns to the start state from which the junction was reached. As the segment in the junctions does not belong to any valid transition, steps 1 – 4 are executed for the transition/segment with the next-lowest priority.
5. If all of the transitions/segments leading away from a state are valid, then no transition occurs and the system remains in the state.

The sequence is represented schematically in Fig. 2-13.

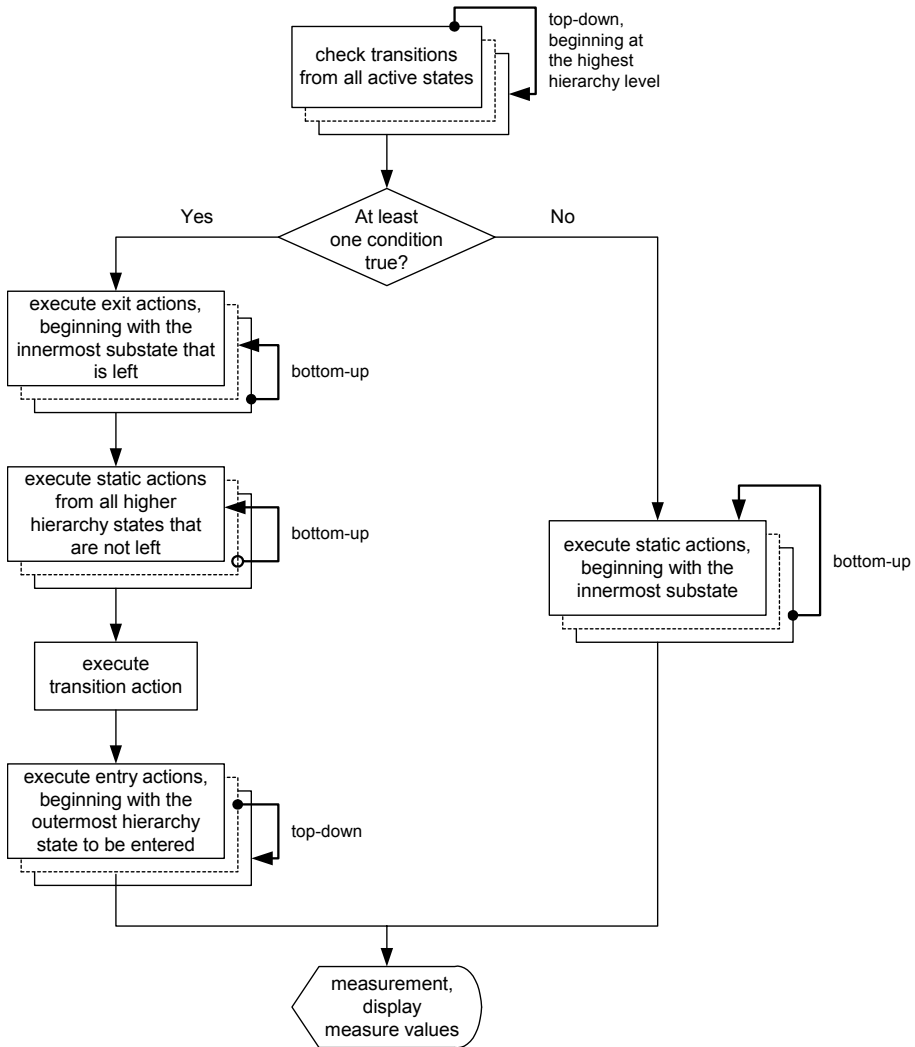
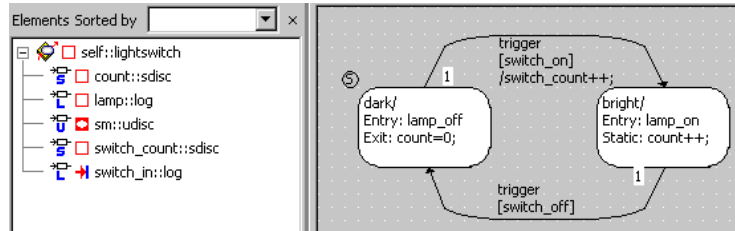


Fig. 2-13 Evaluation of a state machine

2.5.7 Simple Code Example



Parts of the generated code are shown here for this simple state machine.

```

/*
 * -----
 * 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));
}

```

Annotations for the code:

- Exit action from dark: points to `self->count->val = (sint32)0;`
- Transition from dark to bright: points to the `if (LIGHTSWITCH_IMPL_switch_on (self))` block.
- Transition action: points to `self->switch_count->val++;`
- Entry action from bright: points to `self->sm->val = bright;`
- Transition from bright to dark: points to the `if (LIGHTSWITCH_IMPL_switch_off (self))` block.
- Entry action from dark: points to `self->lamp->val = (uint8>false;`
- Static action from bright: points to `self->count->val++;`
- Public method for the output lamp: points to the `LIGHTSWITCH_IMPL_getlamp` function.
- Public method for the input switch_in: points to the `LIGHTSWITCH_IMPL_setswitch_in` function.

2.5.8 Optimizing the State Machine

Usually, there are several ways to specify the same functionality or to adjust the code generation/build process settings.

When code is generated for a state machine, parts of actions and conditions specified at the state or transition are either inserted on the spot (*inlining*) or—on certain conditions—generated as separate methods (*outlining*). The prerequisites for outlining are:

1. The state machine optimization option **Outline Generated Methods (may be changed locally)** is activated in the "Project Properties" window, "Statemachine" node, of the project that contains the state machine.

This options applies to all state machines contained in the project, and to all experiments (physical, quantized, implemented).

2. The option **Outline automatically generated methods for State Machines** is activated in the implementation editor of the state machine.

Note

*When the first prerequisite is not met, outlining is **not** done for any state machine in the project.*

*When the first prerequisite is met, but not the second, outlining is **not** done for **this** particular state machine.*

If both prerequisites are met, code size with and without outlining is checked during code generation. If code with outlining is smaller, outlining is done.

If actions and conditions (or parts thereof) are specified in separate diagrams, the corresponding code is either generated in separate private methods (outlining), or it is inserted on the spot automatically during code generation (*auto-inlining*).

The following prerequisites must be met so that auto-inlining can take place:

1. The state machine optimization option **Auto-inline private methods (Smaller code-size - may be changed locally)** is activated in the "Project Properties" window, "Statemachine" node, of the project that contains the state machine.

This options applies to all state machines contained in the project, and to all experiments (physical, quantized, implemented).

- The option **Auto-inline private methods (Smaller code-size)** is activated in the implementation editor of the state machine.

Note

When the first prerequisite is not met, auto-inlining is **not** done for any state machine in the project.

When the first prerequisite is met, but not the second, auto-inlining is **not** done for **this** particular state machine.

If both prerequisites are met, code size with and without auto-inlining is checked during code generation. If code with auto-inlining is smaller, auto-inlining is selected. This is usually the case for small private functions, or for functions with only a few calls. Each function is checked separately, so that only those functions are inlined whose inlining saves code size.

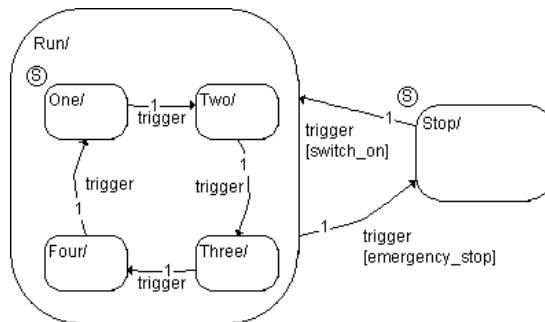
Depending on the possibilities you choose, you can optimize a state machine under three aspects:

- Response time
- Runtime
- Code size

Optimized for Response Time

If response time is the most important criterion, take advantage of the hierarchical structure and the transition priorities. Speed-critical actions are best built into the highest possible hierarchical level to produce efficient code and the quickest possible reaction.

This is illustrated by an example:



If the emergency stop button is pressed (`emergency_stop = true`), the system should stop as fast as possible, i.e. reach the `STOP` state. By drawing the associated transition from the `Run` hierarchy state to the `STOP` state, the transition has the highest priority in the hierarchy and is evaluated first.

If any of the substates is active and the emergency button is pressed, the transition from `Run` to `STOP` is always evaluated and the transition occurs.

Direct transitions from each of the substates to `STOP` are as efficient regarding time, but they require higher maintenance effort because four transitions are specified instead of one.

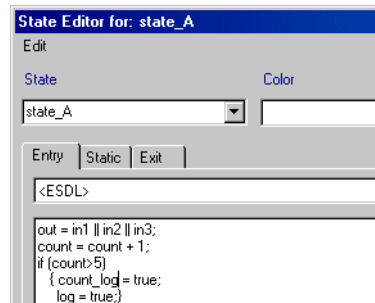
A separate trigger for time-critical events (`emergency_stop = true` in the example) also optimizes response time. The drawback is additional program code for the separate trigger.

Optimized for Runtime

If the total runtime is the most important criterion, you can use several optimization possibilities, individually or in combination, to generate efficient code.

Actions/Conditions: If *actions* or *conditions* are specified with partly or totally the same functionality, this can be done either runtime-optimized or size-optimized. *Runtime-optimized* means that the code for each action and condition is inserted on the spot during code generation. No additional function call is required. The disadvantage is the repeatedly generated code and thus increased memory requirement.

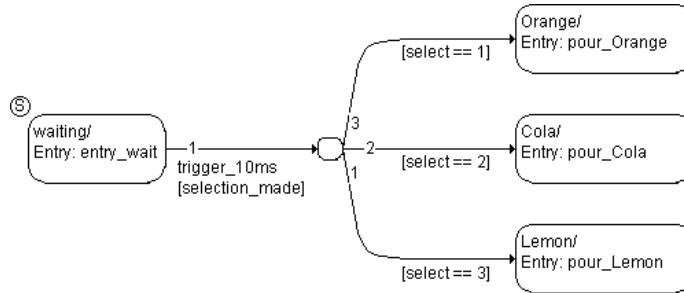
This can be achieved by specifying the code explicitly at the state or condition and deactivating the options **Outline Generated Methods (may be changed locally)** and **Outline automatically generated methods for State Machines** (see prerequisites for outlining on page 73).



The optimization becomes even more effective if auto-inlining (see page 73) is activated. In that case, even actions/conditions specified in separate diagrams are inserted on the spot, if applicable.

With the **Inline** option in the implementation editor of an action/condition specified in a separate diagram, you can enforce *inlining*.

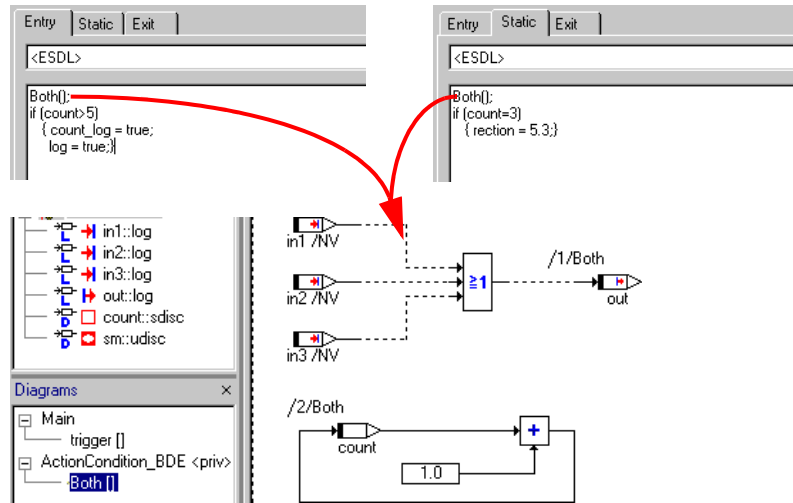
Junctions: If several transitions with partially identical conditions lead away from a state, the use of junctions can bring runtime savings. Identical sections of the conditions are assigned to the transition segment from the start state in the first junction. If these are not fulfilled, the other segments are not evaluated.



Optimized for Code Size

Actions/Conditions: Optimizing actions or conditions *for code size* means that identical parts of actions/conditions are generated as separate private functions that are called at need.

This can be achieved by specifying the repeatedly used parts as methods in a separate diagram, which are then called from the actions (see figure).



As an alternative, you can enter the code directly at the state or transition and use the outlining functionality.

For both alternatives, the code is generated only once. The price to be paid are additional function calls.

In some cases (small private functions, few calls), it may be advantageous, regarding code size, to insert the code on the spot. You can activate auto-inlining (cf. page 73) with the **Auto-inline private methods (Smaller code-size - may be changed locally)** and **Auto-inline private methods (Smaller code-size)** options; with that, you have selected the most effective optimization of actions and conditions for code size.

Static actions of hierarchy states: For static actions in hierarchy states, an additional optimization option exists.

By default, code for the static action of a hierarchy state is generated for each transition that does not lead out of the hierarchy, as well as once for each substate of the hierarchy. In large hierarchies, this can result in a noticeable part of the entire code.

When you activate the **Optimize Static Actions (Restricted Modeling)** code optimization option in the project that contains the state machine, code for the static action of a hierarchy state is generated only once for each substate. Thus, code size can be reduced.

A disadvantage of this optimization is that it does not work for some models. If a state machine contains a substate with a direct transition out of its hierarchy state, this transition must have the highest priority of all transitions from that substate. Otherwise, code generation aborts with the following error message:

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

The changes in code generation change the state machine semantics as follows:

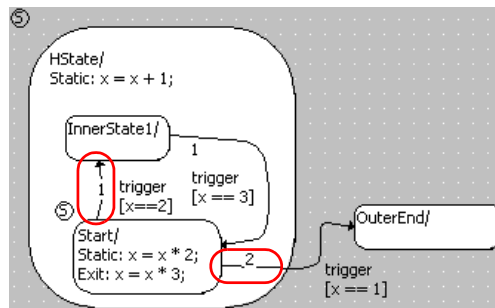
Note

The changes can alter the behavior of the state machine. If you activate the option for an existing state machine, check its behavior carefully.

- The static action of the hierarchy state is executed *before* the conditions of the transitions from the substate are evaluated.
- If no transition occurs, the static action of the hierarchy state is executed *before* the static action of the substate.
- If a transition occurs, the static action of the hierarchy state is executed *before* the exit action of the substate.

Two examples illustrate the effect of this optimization. In both examples, the state machine consists of the hierarchy state `HState` containing the substates `Start` and `InnerState1`, and the base state `OuterEnd`. Two transitions leave `Start`, one of them (`Start` → `OuterEnd`) also leaves the hierarchy state `HState`.

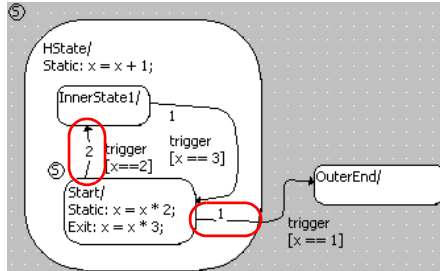
In the first example, the transition from `State` to `OuterEnd` has a higher priority than the transition from `State` to `InnerState1`. This means that code can be generated both with activated and deactivated **Optimize Static Actions (Restricted Modeling)** option.



The following table shows the generated C code for both cases. Code for the static action of HState is set in boldface.

Option deactivated	Option activated
<pre> 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) { x = x + 1.0; sm = Start; return; } x = x + 1.0; return; } </pre>	<pre> 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) { sm = Start; return; } return; } </pre>

In the second example, the transition from State to OuterEnd has a lower priority. With activated **Optimize Static Actions (Restricted Modeling)** option, code cannot be generated.



Option deactivated

```

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

Option activated

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

Hierarchical Code Generation: Two possibilities exist to generate code for a hierarchical state machine:

With *flat code generation*, the hierarchy is flattened, i.e. a single `switch` statement is generated for all (basis) states and transitions.

With *hierarchical code generation*, several `switch` statements are generated, nested according to the hierarchy. To activate this kind of code generation, the following options must be activated:

1. project settings, "statemachine" node: **Hierarchical Code Generation (may be changed locally)**
2. implementation editor of the state machine, "Settings" tab: **Hierarchical code generation for State Machines**

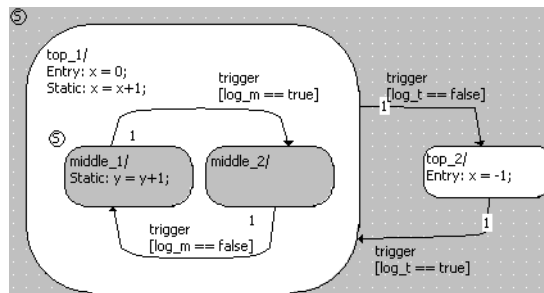
When the first option is not activated, no hierarchical code generation is done. When the first option is activated, the second option activates/deactivates hierarchical code generation for a particular state machine.

With hierarchical code generation, code for transitions from hierarchy states is generated only once, instead of once for each affected basis state with flat code generation. Thus, code size is reduced. The reduction can be considerable (up to 30%). In the experiment, hierarchical and flat code generation behave identical for identical state machines.

Note

For hierarchy states **without** transitions and/or static actions, code size is not reduced, but slightly (1–2%) **increased**.

An example illustrates the difference in the generated code.



Note

The reduced code size does not show in the generated C file, but in the generated executable file.

The transition from top_1 to top_2 is set in boldface.

hierarchical code generation	flat code generation
<pre> 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) { 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; } } } </pre>	<pre> 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 = 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; } } </pre>

```

        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 top_2 :
{
    if (self->log_t->val)
    {
        self->x->val = 0.0;
        self->sm->val = middle_1;
        return;
    }
    return;
}
}
}
}
}
}

```

Triggers and trigger arguments: If *trigger arguments* are used for communication with other ASCET components, instead of inputs and outputs, the static RAM requirements are reduced. You can find more information on this in the next chapter.

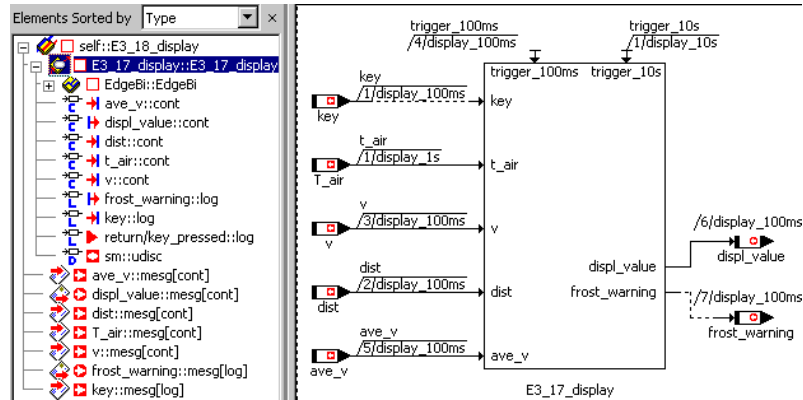
2.5.9 State Machines as Classes

A state machine is a class with special description means. The trigger, condition and actions are modelled as special methods:

- A trigger is a public method without a return value. The state machine is executed whenever a trigger is started.
- A condition is a private method with a return value of type logical.
- An action is a private method. An action has, as standard, no arguments and no return value.

If necessary, you can add arguments to any of these methods, for communication with other ASCET components.

Inputs and outputs serve for the integration of the state machine with other components. The input values are buffered to internal variables and can therefore be used in all computations of the state machine (in contrast to arguments of a method, that can only be used in the method itself). The outputs are also buffered, so they can be read without invoking the computation of the state machine. Each input and output needs its own sequence call (see section 6.3).



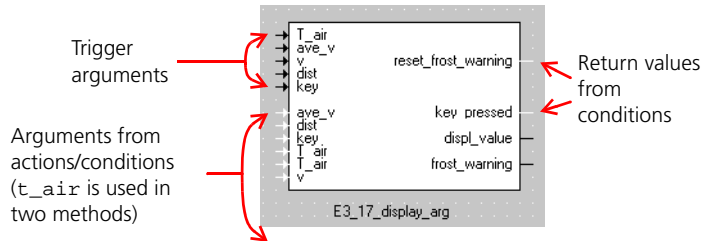
This type of external communication is, however, memory intensive as a variable must be reserved in the RAM for each input and output. To reduce the static RAM requirement, you can add arguments to the triggers (and to arguments and conditions, if these are specified in a separate diagram). You can then use these for external communication. Stack variables which do not burden the static RAM are created for the arguments of a C function. The dynamic RAM area is burdened temporarily.

You should always keep the following points in mind:

- Triggers are public methods. Their arguments can be described outside of the state machines. In the Layout Editor, the trigger arguments are represented by black argument connections.

- Arguments and conditions are private methods. Their arguments are therefore not available outside the state machine. In the Layout Editor, they are represented by white argument connections.

If a trigger argument is to be used in an action or condition specified as a block diagram, an argument of the same type and the same name as the trigger argument must be added to each corresponding method.



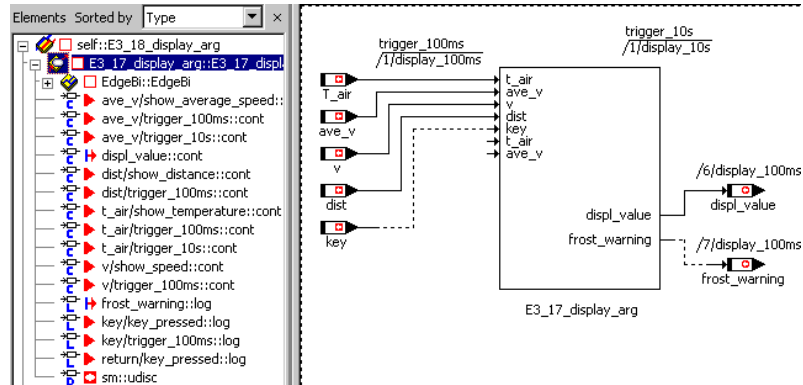
The arguments are depicted according to their name and their type. If, in the trigger and the action/condition, there are arguments with the same names but with different types, a warning is issued. If the argument is only defined in an action or a condition but not in the opening trigger, an error message is output.

Also, there are the following rules for the use of trigger arguments in actions and conditions:

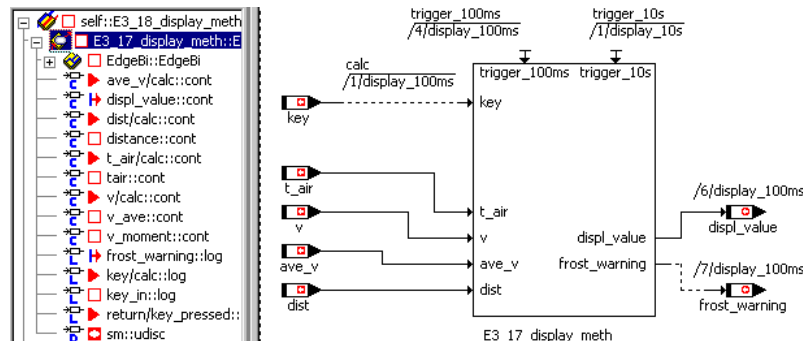
- All trigger arguments which are to be used in the *entry action* of a state must be defined in the action and in every trigger belonging to the transition leading into the state, as it will be started by these triggers.
- All trigger arguments which are to be used in the *exit action* of a state must be defined in the action and in every trigger belonging to the transition leading out of the state, as it will be started by these triggers.
- All trigger arguments which are to be used in the *static action* of a state must be defined in the action and in each trigger of the state machine. Each trigger event which does not cause a transition from the active state, starts the execution of its static action.
- You must define all trigger arguments which are to be used in the *condition* or *transition action* of a transition, in the action/condition and in the triggers belonging to the transition, as they will be started by this trigger.

If one of these rules is violated, an error message is issued.

After the integration into another components, you can assign values to the trigger arguments. In contrast to the inputs and outputs, only a sequence start (see section 6.3) is required for all the arguments.



Additionally, 'normal' public methods can be defined for a state machine like with all other classes. These public methods offer several additional possibilities. They can be started from outside the state machine, e.g., as well as from the states and transitions. Their arguments and return values can replace inputs and outputs in the communication with other components. In this case too, only a single sequence start is required for the complete method (this does not, however, bring any runtime savings). You also have the option of preparing the input values, should they be needed in the state machine.



Further applications of public methods in state machines are, for example, reset functions that can be started both from within and without the state machine, or counters that have to register events inside and outside the state machine. Parts of the state machine, integrated classes, can be calculated in a

different time frame. You can integrate a second state machine into the first and - without an additional trigger - computed in a different time frame, too, by starting it via a public method.

3 Types and Elements

Every algorithm in a component works on *elements*. An element contains a piece of data, and makes available an interface for accessing its data or returning the value of a computation (e.g. interpolation of a characteristic line). Elements are strongly typed, i.e. each element is of a fixed type. Since there can be more than just a single element of a given type, an element is referred to as an *instance* of a given type.

ASCET has a number of basic types, that can be used directly, such as discrete or continuous variables, arrays, matrices or characteristic lines and fields. New, user-defined types can be added to the system in the form of classes. Classes are complex types, they have a complex structure, because they are usually build up from other types (basic as well as other complex ones). The types can be classified as in the following diagram:

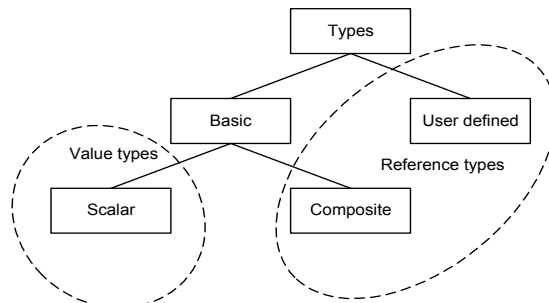


Fig. 3-1 Classification of data types in ASCET

As the modelling in ASCET takes place on the physical level, the types are also 'physical' types. Elements are committed to a specific data type (e.g. `unsigned int8`) only during the implementation phase, which is independent of the modelling phase.

The physical definition of an element must contain the following information:

- the name of the element
- the model type
- the element kind
- the scope of the element

The options that are available for each of the above categories are described in detail in the following sections.





When defining an element, additional information on the physical unit and a comment can be added to generate a meaningful documentation of the model. This information has no impact on the physical model.

3.1 Basic Model Types

In ASCET there are two categories of basic model types: scalar types and composite types.

3.1.1 Scalar Types

The most important of the basic model types are the scalar ones. ASCET supports four basic scalar types, which are represented in the various ASCET windows by their respective symbols:

-  • *Continuous* is used for continuous physical values that can be infinitely large and have an arbitrarily fine resolution. This type is suitable for modelling variables like temperature, speed, etc.; it is referred to as model type `cont`.
-  • *Signed discrete* is used to model integral numbers of arbitrary size; it is referred to as model type `sdisc`.
-  • *Unsigned discrete* is used to model non-negative integral numbers of any size. This type is suitable for modelling things like the number of cylinders of an engine; it is referred to as model type `udisc`.
-  • *Logical* is used to model logical information, e.g. whether a particular system is active or not; it is referred to as model type `log`.

The four basic scalar types are value types. Whenever an element of such a type is used, not the element itself as an object, but its value is used. Automatic typecasting between the arithmetic types `cont`, `sdisc` and `udisc` is performed if necessary.

Like complex types (classes), each basic type has an interface, i.e. methods to access it. For the basic model types these methods are fixed, the interface cannot be modified.






Scalar types have two simple access methods for the value stored in an element of the basic scalar type, i.e. for writing a new value to and reading the current value from the element:

- `set (type a)`: This method takes one value, e.g. the value `a`, and overwrites the value of the element with that value. If the type of the value does not fit to the type of the element, a type conversion is performed automatically.
- `get ()`: This method returns the current value of the element. The value returned is of the same type as the element itself

Accessory methods in basic types are invoked automatically, when an element name is used in an expression or when an assignment is performed. They do not have to be coded explicitly.

3.1.2 Composite Types

Composite types are basic types that are built up from basic scalar types. The following composite types are available in ASCET:

- array ()
- matrix ()
- characteristic line ()
- characteristic map ()
- distribution ()

Composite types consist of basic scalar types. Arrays and matrices can consist of all four scalar types, characteristic lines, maps, and distributions only of the three arithmetic types. Unlike basic scalar types, composite types are *reference types*. When assigning two variables of reference types to each other, not the values are assigned (and copied), but the references to the variable.

All reference types have access methods for their elements:

- `set (reference type a)`: This is an assignment of the reference to reference type a. After such an assignment, both elements (the assigned as well as the assigning) are the identical element!
- `get`: This returns a reference to the element of composite type.

Parameter passing in method calls works in the same manner as assignments. A reference is passed to the element. As a consequence, a change to the parameter, for instance by assigning a value to it, is also reflected outside the method. This mechanism is equivalent to a "call by reference" in programming languages like C.

Array



An array is a basic type, holding a number of scalar values of the same basic scalar type, e.g. `continuous` or `logical`. The position of a scalar value within an array is indicated by its associated index value which must be of the model type `unsigned discrete`. The size of an array is limited to 2048, and must be defined statically. The array index takes values between 0 and `size-1`.

The interface of an array consists of the following methods:

- `void setAt(scalar type a, udisc i)`: The assignment of the scalar value `a` to the position `i` in the array.
- `scalar type getAt(udisc i)`: Returns the value at position `i` of the array.

Arrays of non-scalar basic types or complex (user-defined) types are not available.

Matrix



A matrix is similar to an array. A matrix is two-dimensional, however, so it takes two indices. The type of index is the same as that of an array (udisc). The size for each dimension is limited to 63, i.e. the indices take values between 0 and 62.

The interface of an array consists of the following methods:

- `void setAt(scalar type a, udisc i, udisc j)`: The assignment of the scalar value `a` to the position `(i, j)` in the matrix.
- `type getAt(udisc i, udisc j)`: Returns the value at position `(i, j)` of the matrix.

Matrices of non-scalar basic types or user-defined types are not available.

Characteristic Tables



To support nonlinear control engineering, one-dimensional and two-dimensional characteristic tables are available in ASCET. The former are called *characteristic lines*, the latter are called *characteristic maps*. Characteristic tables are used to describe a value in dependence of one or two other values, where either the functional dependence is not known exactly or calculating the function would be computationally expensive.

An example for a characteristic line is the throughput of a diode in dependence of the input voltage. This characteristic behavior is described by a curve. The curve is represented as a table of sample points, each of which is associated with a sample value. The sample points represent the x-axis of a function graph, the sample values represent the curve being described.

Accordingly, a characteristic map is represented by a two-dimensional table of sample points for pairs of input values, where a sample value is associated with each pair of sample points. The size of characteristic tables is limited to 2048

sample points for characteristic lines, or 63 sample points on each axis for characteristic tables. Characteristic tables are always parameters, i.e. they can only be read from within the model.

Each characteristic table is also associated a interpolation and extrapolation routine. These routines determine, how the output value of a characteristic curve is determined by the input value(s).

ASCET provides two different interpolation modes: with rounded interpolation the value between two sample points is derived from the sample value at the lower (left) sample point, with linear interpolation the value is derived from a straight line between the sample values.

In controller applications interpolation is a very time consuming operation. It consists of two operations: searching for the right interval of sample points and calculating the interpolation factors, and secondly, calculating the output value from the interpolation factors.

The computation of interpolation factors can be optimized using two special types of characteristic tables in ASCET: group tables and fixed tables. *Group tables* do not contain a sample point distribution, but reference a distribution of sample points. Distributions can be shared by many group tables. The computation of the interpolation factors is performed only once for the distribution, and only the computation of the output value is performed for each group table separately.

A *distribution* is always a one-dimensional table of sample points. Two-dimensional group tables therefore reference two distributions.

Fixed tables have a equidistant distribution, i.e. the sample points have a constant distance from each other. This makes the computation of interpolation factors much faster. The memory requirements are lower as well, since instead of a list of sample points, only an offset and a distance have to be stored. There is, however, no combination of fixed and group tables.

The interface of a characteristic table depends on its dimension and whether it is a normal, fixed or group table. There are basically three methods:

- `void search` (arithmetic type a): This method applies to the distribution of a characteristic line. Here the correct supporting points are searched, and the interpolation factors are computed. For two-dimensional tables there are two parameters, i.e. `void search` (arithmetic type a, arithmetic type b).
- arithmetic type `interpolate()`: This method interpolates the value of the characteristic line or map from the interpolation factors and the value points at the associated supporting points.

- arithmetic type `getAt` (arithmetic type `a`) is the combination of the search and interpolate method. For two-dimensional tables, there are two parameters, i.e. `void getAt` (arithmetic type `a`, arithmetic type `b`).

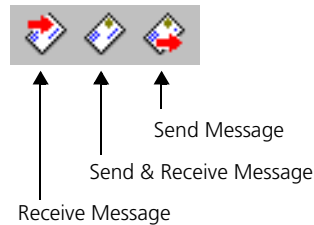
For one table, the parameter and the output value must be of the same arithmetic type, e.g. there is no characteristic map where continuous and discrete types can be mixed. The separation of the method `getAt` into the methods `search` and `interpolate` only makes sense for group tables.

A distribution only has the method `search`. A group table only has the method `interpolate`. A regular or fixed characteristic table has all three methods.




3.1.3 Real-time Language Constructs

ASCET provides a number of language constructs for real-time applications in the description of components.

Messages




Messages form the input and output variables of processes and are used for interprocess communication in the same way as basic scalar types. Unlike global variables, messages are *protected variables* in preemptive scheduling. If two concurrent processes both access the same message, data consistency is guaranteed, because each process works on its own copy. Messages are only available in modules. Depending on their usage, there are three different types of messages:

- *Receive messages* can only be read. Receive messages are used as inputs to a module. 
- *Send messages* can only be written to. They are used for the results of the computations of a module. 
- *Send & Receive messages* can be read from and written to. 

Resources



A resource (type symbol ) represents a part of an application that can only be used exclusively, e.g. timers or special devices. In order to access a resource, there are two methods:


- `void reserve()`: the resource is reserved, that is the access to it is blocked.
- `void release()`: the resource is released, that is access to it is granted again.

By executing the `reserve` method, access to the resource is blocked and exclusive access is guaranteed in a preemptive environment, i.e. if the current process is de-scheduled and another process wants to use the resource, the access is denied.

When access to the resource is no longer required, the resource can be released by the `release` method. This makes the resource accessible to other components again. To avoid deadlocks or priority inversions, the reservation of a resource is linked to the priority ceiling of the corresponding process. Resources are always global elements.

The ΔT Parameter



In control engineering applications the result of the calculations within a component often depends on the value of the sampling rate. ASCET provides the system parameter ΔT (type symbol ) for uniformly describing the algorithms for all sampling rates. The value of this parameter is provided by the operating system and represents the time difference since the last activation of the currently active task.

Note


The name ΔT is reserved for the system parameter. You can create no other element with that name; since reserved keywords so not distinguish between upper and lower case, ΔT , Δt , and ΔT are reserved, too.

3.1.4 Special Types

Several other types exist besides those already described. They are discussed here.

Enumeration



Enumerations (type symbol ) are unique types with values taken from a group of known constants called enumerators.

Literals



Literals are strings that represent a fixed value of a basic scalar type which can be used in any expression. The value of a literal is either a number (discrete or continuous), a character string, or one of the values `true` or `false` (logical). In the block diagram editor the values `string`, `true`, `false`, `0.0`, and `1.0` are predefined.

3.2 The Kind of Elements

Each element has a kind. The kind of an element describes how the element is used, either as a variable, a parameter, a system constant or constant. Implementation-Casts are another kind.

- *Variables* store values that can be read and written from inside the model, i.e. a read and a write operation can be performed on them.
In the ECU, they can be placed in the *volatile* or *non-volatile* memory. For newly created variables, *volatile* is pre-selected.
- *Parameters* store values that can only be read from inside the model. Parameters can also be calibrated, i.e. written to from outside the model. In some cases, special prerequisites are required for that purpose, e.g., the connection to a calibration tool.
Parameters (including characteristic lines/maps) are automatically set to *non-volatile*; in the ECU, they are placed in the respective memory.
- *Constants* store values that can only be read from inside the model. In contrast to parameters, constants cannot be changed from outside the model but are fixed at specification time. Constants cannot be implemented, either.
Constants are created as a `define` statement in the generated C code. However, they are not necessarily explicitly visible in the generated code. If, e.g., the constant is set against a requantization, the constant does not explicitly appear.

- *System constants* are used like constants, and also created as `define` statements. Unlike constants, system constants can be implemented. They are always explicitly visible in the generated code.

System constants can be converted into normal constants using [Extras → Convert System Constants to Constants](#) in the Component Manager.

Tab. 3-1 summarizes the differences in usage between variables, parameters and constants.


















- *Implementation casts* (see section 4.2.4) provide the user with the ability to specify the implementation in a targeted manner at any chosen position of a calculation or a data stream. Unlike variables and parameters, implementation casts do not allocate any memory, and thus have no storing effect in the model and cannot be calibrated.

Implementation casts do not have data; they are always of the `cont` model type, always have a scalar dimension and a *local* range of validity (see section 3.3). Unlike other elements, the properties of implementation casts cannot be edited.

	Model	Experiment / Calibration Tool	Implementation
variable	r-w	r-w	yes
parameter	r	r-w	yes
system constant	r	r	yes
constant	r	r	no
implementation cast	—	—	yes

Tab. 3-1 Synopsis: variable, parameter, system constant, constant, implementation cast

The kinds of elements are marked by certain symbols in various ASCET windows (e.g., field "3 Contents" of the Component Manager).

	Scope				
	imported	exported	local	dependent	virtual
variables ^a					 ^b
messages					
parameters ^c					^d
(system) constants					
implementation casts					
dT					

a: including arrays, matrices and enumerations

b: independent of scope; see page 99

c: including characteristic line/map, distribution

d: symbol is derived from other settings (scope, etc.)

Tab. 3-2 Symbols for the various element kinds and scopes

Temporary Variables

To avoid multiple execution within the same method or process, temporary variables can be specified for each operator or method call. With that, the value of the expression is computed only once for each method or process it is used in, and stored to a temporary variable. When the expression is used again in that method, it is not re-evaluated but the temporary variable is reused.

Each specification editor can create a temporary variable. A temporary variable does not have a start value; its value is determined only by the assignment of an expression. ASCET internally manages the temporary variables and provides a unique assignment (e.g. in the branches of an `IF` statement) so that no undefined values turn up when the temporary variable is used later. The value remains valid until a new assignment to the temporary variable occurs.

The example shows the temporary variable `t` which stores and reuses the value of the addition `a + b`:

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

Virtual Variables/Parameters

Virtual variables/parameters are only available in the specification platform, they bear no relevance for code generation. They are included for a better understanding of the significance of model elements in the specification.

Virtual variables always depend on other virtual or non-virtual variables. Virtual variables are merely aliases to non-virtual variables. No mathematical dependencies such as formulae are allowed; thus the identity (`var_virtual = var_real`) is predefined for editing the data of virtual variables.

On the other hand, parameters declared as virtual are not necessarily dependent on other parameters.

Dependent Parameters

Model parameters can be connected to other system or model parameters via a mathematical dependency. Calibrating parameters can therefore lead to inconsistencies.

To avoid possible inconsistencies from parameter calibration, it is possible within ASCET to specify the dependency of a parameter in the specification editors. The dependency of a parameter is represented by a mathematical formula.

Note

*Dependent variables do **not** exist.*

3.3 The Scope of Elements

Some elements are used for exchanging data between different components. To establish this, elements can be exported from one component (or from the project) and can be imported in any other component. Here, the matching is done via names. The scope of each element can be defined as one of the following:

- *Local* elements can only be used within the component that defines them, i.e. in all methods or processes of that component.
- *Imported* elements are defined in some other component or project, but can be used in the component that imports them. The properties of an imported element can be changed only in the context of the component that defines and exports the element.
- *Exported* elements are defined in one component and can be accessed by all other components by importing that element.

- *Method/Process-local* elements can only be used in the method/process that define them. Method/Process-local elements are not static and do not have a data set.

3.4 User-defined Model Types

Elements can also be user-defined model types, i.e. modules or classes. User-defined model types are always reference types. The interface is defined by the interface of this component.

The scope of a user-defined type can be the same as that of the basic types, namely imported, exported, local and method-local. Like arguments, method/process-local elements of a reference type are not instantiated, but a reference to them is established. This means that, when using a method/process-local element of a reference type, an assignment to this element must precede any further use of that element.

The kind of an element is irrelevant for user-defined model types. User-defined model types are always treated as variables, i.e. there is no restriction of the interface from within the model.

4 Data and Implementations

In the previous chapter the parts of a component were identified as the set of elements, the interface of the component, and the functional description of the methods or processes in the form of algorithms.

In this chapter two additional parts of a component specification are introduced: data and implementation. Both data and implementation belong to the elements in a component, i.e. both describe properties of the elements.

The approach of separate descriptions for data and implementations is not usually found in standard programming languages, where the data assignments of variables is part of the functional specification, i.e. the program code.

The data of a component describes the physical values with which the elements of the components are initialized. Data contain physical information and are thus part of the physical specification of the component.

Also, standard programming languages do not usually separate between the implementation of a functional specification and the functional specification itself. The functional specification is usually identical to its implementation.

4.1 Data

The data of a component describes how the elements of a component are to be initialized. Thus data refers to the elements of a component.

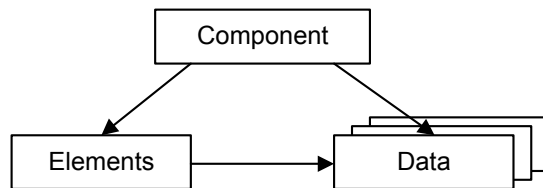


Fig. 4-1 A component with multiple data sets

The data is held separately from the elements because a component can have multiple instances in a project, whereas the different instances access different data sets for their elements. (The data sets are, however, *not* parts of the respective instance.)

An example would be a p-control filter. Each instance of this p-control filter has its own value for the p-factor. This is achieved by assigning different data sets to the p-control.

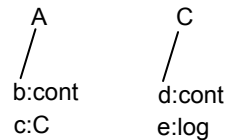
The specification of data is part of the specification of the component itself, and not of the different instances. This may lead to a large number of different data sets for a component, but if each instance would hold its own data, this would result in the loss of a modular system design.

The organization of data for each element depends on whether it is a basic or complex element. Since basic elements are always used within complex objects, and are never considered separately from those, basic elements do not have explicit data sets. The data for the basic elements are therefore part of the data set of the complex element they are contained in.

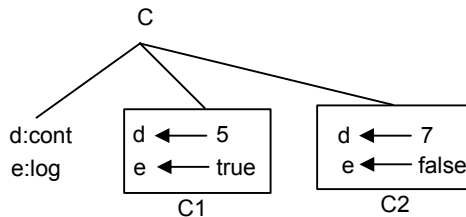
Complex elements are the components specified by the user. Each complex element has its own data set. If a complex element is used in a component, the data set of the complex element is referenced by the component. Thus the data of a component has the same hierarchical structure as the component itself.

Data sets have an object ID, which is used to reference the data of a component. Just like references to user defined types, this reference is not name-based.

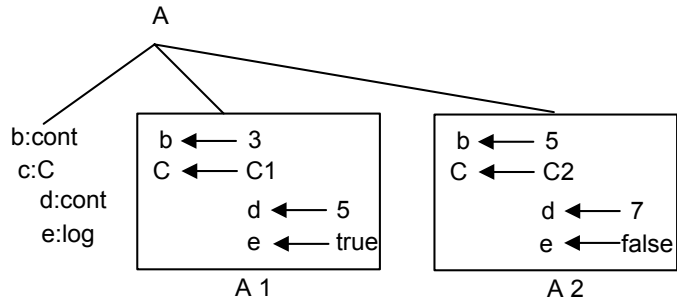
Consider the following example with the types A and C:



The type C has the following data sets:



A data declaration for the type A using the data sets of C would have the following results:



The data for the basic types can be specified directly. For the scalar types the data consists of one value. For composite types, like arrays or characteristic lines, the data consists of a table of values, or a table of sample points and sample values.

4.2 Implementations

Implementations describe how the elements of a component are to be realized in code. Here the same scheme as for data is followed:

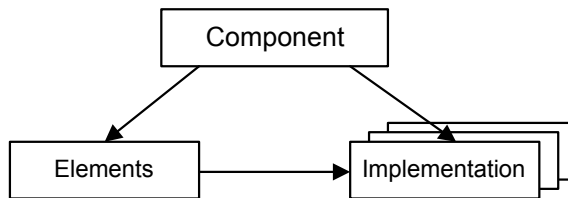


Fig. 4-2 A component with multiple implementations

The same reference scheme applies to implementations as to basic and complex types. The effect of implementations is much broader than that of data sets. The implementation of an element, e.g. whether an element of type `cont` is represented as a data type `float` or `signed int`, has direct influence on the code that is generated from the functional description for a method or process.

4.2.1 Implementations for Scalar Types

The implementation describes how an element of a basic type is realized in the generated C code. The implementation specification for elements of type logical is very easy, since a logical element has only two values, either true or false.

The implementation specification consists only of the data type. For logical elements either `byte`, `word`, or `long` can be chosen.

The implementation specification for the arithmetic types is much more complex. It describes, among other things, the implementation type, which can be an integer type even for elements of type `continuous`. The implementation specification therefore contains a complex transformation from the physical domain to the implementation domain, which can be very different from each other.

The differences between the physical domain (e.g. model type `continuous`) and the implementation domain are the infinite range of the physical domain from $-\infty$ to $+\infty$, and its arbitrarily fine resolution. In the implementation domain, on the other hand, the range is limited by the word length, and the resolution is not arbitrarily fine but fixed to 1.

In order to make a transformation between the physical domain and the implementation domain possible, the range of the physical domain has to be limited. Thus each element must be assigned an interval for the relevant physical values. The resolution must also be restricted. Therefore, each element has to be given a fixed resolution, the quantization.

For example, let A be a range of values in the physical domain, $A = [-1, 0.5]$, and assume a quantization of $q = 0.2$.

The result of the limitation of the range to an interval and of the quantization is a restriction of the values of an element to a finite set of equidistant values.

$Aq = \{-1, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4\}$

This finite set of values can now be mapped to an integer range:

$Aint = \{-5, -4, -3, -2, -1, 0, 1, 2\}$

This corresponds to a linear conversion formula between the physical domain to the implementation domain of the kind $impl = 5 * phys$. The data type for the integer variable is automatically determined from the integer range. In this example, the data type `signed int8` would be chosen.

When the range of the physical element has an offset larger than zero, the associated integer interval may only contain a few values, but a large data type has to be used.

Consider for example the physical domain range $A = [120, 130]$ and a quantization of $q = 0.5$. A linear conversion would result in an integer range $Aint = \{240, \dots, 260\}$.

The type for the integer variable is `unsigned int16` in this case, although the number of values would also fit into a variable of type `int8`.

To implement this, a general linear conversion formula with an offset can be specified. In the above example, a conversion formula of the type

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

would lead to an integer interval of $\{0, \dots, 20\}$ and a variable of data type unsigned `int8` would be sufficient.

The conversion formulas are not specified in the context of a component, but in the context of a project. This makes it easy for several components to use the same conversion formulas. Furthermore, this complies with the ASAM-MCD-2MC standard.

4.2.2 The Implementation of Composite Types

For composite types like arrays, matrices or characteristic tables, the implementation is specified for the interface elements of the composite types, which themselves are of a scalar type.

For arrays, for instance, the implementation for the elements held in the array must be given. This implementation is valid for both, the input and the output of the array. The implementation for the index is fixed, since the index is a discrete model type.

For characteristic tables, the implementation of the x-points and y-points and the values of the table can be specified separately from each other.

4.2.3 The Implementation of User-Defined Types

The implementation of user-defined types consists of the implementations of all elements used in that component.

In the case of classes, the arguments and return values also need to have an implementation, since the value of an actual and formal argument have to be adjusted correctly to each other. This is automatically done for arguments of a scalar type.

This automatic adjustment does not work for arguments of composite or complex types. If such arguments are used, the implementation of the formal argument and the actual argument must coincide. Here, no automatic adjustment is possible, since these arguments are passed as references.

Temporary elements do not have an explicit implementation, but they are automatically assigned an implementation by the code generation algorithm. It is important that an assignment to this variable (e.g. an initialization) precedes any other use of it.

Method- and process-local elements can be implemented automatically, like temporary elements, but they can be explicitly implemented, too (see ASCET user's guide, section "Implementations of Method- and Process-Local Variables"). The implementation is preserved within the method/process.

4.2.4 Implementation Casts

ASCET 5.0 introduced a new primitive element type – the *implementation cast*. Implementation casts provide the user with the ability to influence the implementation of intermediate results within arithmetic chains. This allow the user to display knowledge regarding particular physical correlations (for example, that a specific range of values is not exceeded at a defined point in the model) in the model, without requiring the allocation of physical memory.

Note

*Implementation casts **cannot** be used in conjunction with logical elements.*

Below is a small example to illustrate this functionality.

In a simple arithmetic specification, two variables, *a* and *b*, are added, the result of the addition is multiplied by the literal 2, and the result of the multiplication is assigned to variable *c*.

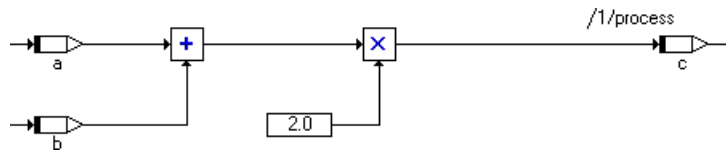


Fig. 4-3 Simple Calculation without an Implementation Cast

During implementation, variables *a*, *b* and *c* have been assigned the `int16` type; all three variables exhaust the entire possible value range. Because of this, the code generator in the example above would create a 32-bit-wide temporary variable, and would requantize this before assigning it to *c* to a value range that is applicative for `int16` by executing a right shift.

Now, if the user knows that the sum of *a* and *b* can be no greater than a 16-bit-wide result and thus exhausts only half of the possible value range (for example, due to physical boundary conditions or because certain correlations in the model compel this to be the case), he or she can define this as such using an implementation cast (see Fig. 4-4).



Fig. 4-4 Simple Calculation with an Implementation Cast

In implementing the implementation cast with the `int16` type and value range `[-16384..16383]`, while disabling both the **Limit to maximum bit length** and **Limit Assignments** options, the user guarantees specific properties of the intermediate result for the code generator. This prevents the requantization required in the example illustrated in Fig. 4-3.

Another application for implementation casts is the targeted allocation of implementations to the inputs and outputs of operators. This function allows you to select target arithmetic services (see section 4.14 "Arithmetic Services" in the ASCET user's guide) for specific operators. In this context, implementation casts replace the present operator implementations.

As the name implies, implementation casts only affect the implementation. More accurately, this means that implementation casts are taken into account for the code generation of experiments (see chapter 4.8.8 in the ASCET user's guide) of these types:

- `implementation experiment` and
- `object based controller implementation`

They are simply ignored for these types:

- `physical experiment` and
- `quantized physical experiment`

Depending on the code generation options (see "To adjust the project settings:" in the ASCET user's guide) for the implementation experiments, implementation casts have the following properties:

- If the maximum bit size that is defined for the project is smaller than 32 bits, the code generation for implementation casts allows the use of a larger bit size. If, however a variable that exceeds the permitted bit size is necessary in the code, an error message is displayed.

With this functionality, implementation casts can be applied within arithmetic chains to specify intermediate results that are outside of the controller's original maximum bit size.

- If an implementation cast is present at the numerator input of a division operator, its implementation overwrites the **Allow Double Bit Size for Division Numerators** option.

Another important property of the implementation cast is that it allocates for its implementation during code generation neither permanent nor temporary memory. This is because implementation casts are not created as global elements or as local function variables. For implementation casts that are applied in combination with a value limitation, however, a local, temporary function variable can be necessary to temporarily store the calculation result before area check is carried out.

The use of implementation cast is limited to the block diagram editor and the ESDL editor. Furthermore, these elements are only offered for modules and classes (excluding, however, CT blocks, Boolean tables and condition tables) and for specifying conditions and actions in state machines.

4.3 Code Generation with Implementations

When choosing an implementation, the code is generated in fixed point arithmetic. This fixed-point arithmetic is based on integer arithmetics. The information of the implementation applies to elements of a component. This information together with the functional description, i.e the information how the elements interact with each other, is the basis for integer code generation.

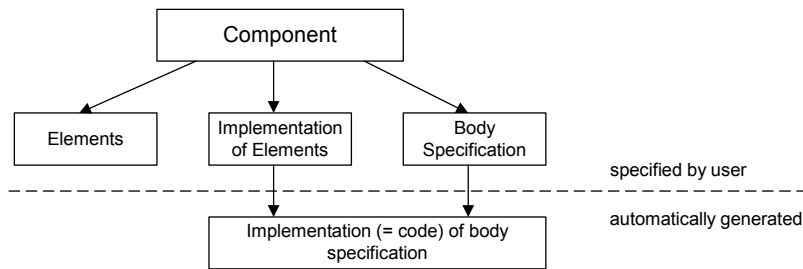


Fig. 4-5 Code generation with implementations

To make the principle of integer code generation more transparent a simple example is given in the following.

An Example: Code Generation for an Addition

Imagine the following simple example

$$c = a + b;$$

where a , b , and c are model variables of type `continuous`.

The implementation transformation is linear without an offset. The following quantizations are used: 0.01 for a , 0.04 for b and 0.05 for c . A , B , and C are the corresponding implementation variables for the elements in the generated C code.

When generating code for the above example, the quantizations must be taken into account. For the values $a = 1$, $b = 0.6$, and consequently $c = 1.6$, the result with the above quantizations would be $A = 100$, $B = 15$ and $C = 32$. A direct transformation of the model to the implementation level would lead to a wrong result ($A+B = 100 + 15 = 115$ which is not equal to $C = 32$).

The reason is that the quantization is not taken into account. The above model equation must be transformed to the implementation transformation. Here the quantizations of A and B have to be adjusted before the addition takes place, and the result of this addition has to be adjusted to the quantization of C. This leads to the following piece of C code for the above model:

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

The multiplication of B by 4 corresponds to the adjustment of the quantization 0.04 to 0.01, and the division by 5 corresponds to the adjustment of the quantization of 0.01 to 0.05.

4.3.1 Transformation of Data under Implementation

The data stored with an element always contains the "model data", i.e. the physical values, but the implementation must also be reflected in the data. In the above example, the physical (model) data for variable a was 1, the data for the implementation variable A however was 100.

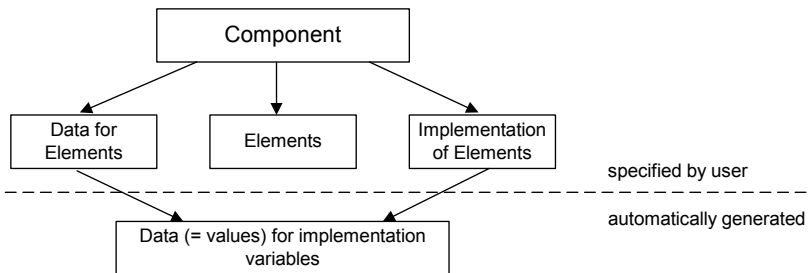


Fig. 4-6 Transformation of data

4.3.2 General Rules for the Implementation Transformation

The implementation transformation works on arithmetic values. The values are adjusted in all arithmetic expressions, so the corresponding arithmetic operations can be executed:

Addition and Subtraction:

The arguments of these operations are adjusted to an quantization. This quantization is determined by the internal code generation algorithms and minimizes the number of re-quantizations. The constant offset is calculated for the result from the quantizations and the offset of the arguments.

Multiplication and Division:

The arguments of these operations are first made offset free, before the multiplication or division can take place. The quantization must not be adapted, but is determined from the result of the multiplication or division. However, to

avoid overflow or a loss in precision, the quantization of the arguments may be multiplied by a power of two (shift operations). This is also automatically determined by the internal code generation algorithm.

Comparison, Minimum and Maximum:

Similarly to addition, the arguments are adjusted to each other (as well in quantization as in offset). The minimum and maximum operator work like the addition operator.

Assignment:

The value that is assigned to a variable is re-quantized and the offset is corrected before assignment is performed. This also applies to argument passing.

4.4 The Implementation of Methods and Processes

The facilities for using implementations (enhanced in ASCET 5.0) allow for method implementations to be specified. Method and process implementations are available in both ESDL and block diagrams.

The implementation of a method or process contains information the memory to be used for running a method or process and whether it should be fully expanded during code generation.

In general, algorithms that should have a short response time or are used more often, will be run in internal memory, whereas other algorithms that are not used very often, such as initialization algorithms, will run in external memory.

In addition, method and process calls can either be represented as function calls or fully expanded in generated code (*inlining*).

5 **Body Specification in ESDL**

This chapter describes the common features of ESDL that are used in the description of classes and modules. The description is divided into three main parts.

The first section contains a brief description of general ESDL characteristics. A comprehensive description of both the syntax and elements of ESDL is provided in subsequent sections.

The differences between ESDL and block diagrams as well as those between ESDL and the C and Java programming languages are summarized at the end of this chapter.

Readers are assumed to be familiar with either the C or Java programming language (or both). If you need further information on C or Java, you can use any of the standard reference manuals for these languages.

The following is a list of some common reference manuals for Java and C:

- Arnold, Ken, Gosling, James, *The Java Programming Language* (Reading, Mass.: Addison Wesley, 1996)
- Flanagan, David, *Java in a Nutshell* (Cambridge, Mass.: O'Reilly, ²1997).
- Kernighan, Brian W., Ritchie, Dennis M., *The C Programming Language* (Englewood Cliffs: Prentice-Hall, ²1988).

5.1 **ESDL as a Modelling Language**

ESDL was designed specifically as a modelling language for the automotive environment. In ASCET, it is used to specify the method or process bodies within classes or modules. For simplicity, classes and modules are subsumed under the term classes in this section.

In ESDL, both the syntax and elements are based on the Java programming language to provide for a low learning curve. When working with ESDL, however, it is important to keep in mind that ESDL is radically different from other languages.

The main characteristics, which in part distinguish ESDL from other languages, are as follows:

- ESDL is a *modelling language*, not a programming language. It is a modeling language that works on the same abstract, physical level of description as the block diagrams commonly used in ASCET. Concepts that are related to or dependent on implementation, such as pointers or shift operators, are not available.

- ESDL is used for systems that run in a *real-time environment*. Hence, it must meet the requirements of real-time operation. As a consequence, ESDL is as object-oriented as these parameters permit. The model structure can be mapped to classes and modules, but instantiation is static and there is no inheritance.
- ESDL is used to build *automotive software*. While users can build complex software models in ESDL, concepts that are currently not relevant to embedded systems, such as string operations, are not implemented.
- ESDL ties in seamlessly with the ASCET development environment. The language is used at the same level as block diagrams, that is, for describing the functions contained in *method or process bodies*. Import of elements and variable declaration are performed using the corresponding tools in the ESDL editor.

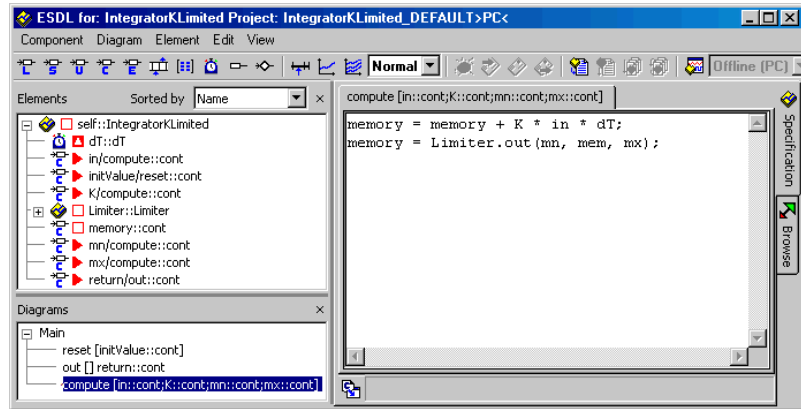
These four main characteristics of ESDL determine the scope and usage of the language. Otherwise ESDL can—more or less—be seen as a highly specialized variant of the Java programming language.

5.2 Basic Elements

5.2.1 Working with Methods and Processes

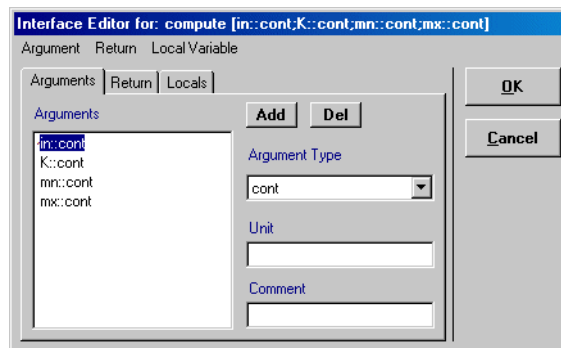
The basic elements of a functional description in ESDL are methods and processes. A method consists of a method header, which serves as an identifier, and the method body which describes the operations to be performed.

The method header consists of the method name, a list of arguments and a return value. Method names are assigned when adding a new item to the methods list ("Diagrams" pane) of the ESDL Editor. They can be modified by renaming the list item.



Method names must be unique in ESDL. Method *overloading* is not supported, i.e. it is not possible for two methods to differ from each other only in the number of parameters and/or parameter types.

The arguments and the return value are optional elements of the method interface. The method header and interface can be modified using the Interface Editor on the ESDL Editor window. The Interface Editor is used to add or modify parameters and the return value as needed.



The functional description of a model is contained in the method body which can be edited in the text pane of the ESDL Editor.

5.2.2 ESDL Syntax

ESDL syntax is entirely the same as that of the Java programming language. Every statement in ESDL is terminated by a semicolon (;).

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

Compound statements or blocks are contained in curly braces { ... }.

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

Method parameters and expressions are contained in parentheses (...).

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

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

The equals sign (=) is used for assignments.

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

5.2.3 Variable Names

In ESDL, variable names are made up of letters and digits. The first element of a variable name must be a letter. The underscore character counts as a letter. Variable names must not contain spaces.

The following are valid ESDL variable names:

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

The names of all variables must be unique within the scope of the current element. This limitation is important when working with imported classes or modules. ESDL does not, at this stage, resolve name conflicts.

Reserved Keywords:

The following keywords are reserved and may not be used as variable names.

```
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.
```

Since upper and lower case are not distinguished, *any* spelling of the above names is reserved.

5.2.4 Data Types

ESDL is strongly typed and variables must be declared. The procedure here is the same as when editing block diagrams. Variables are added to the elements list and can then be edited as needed.

There are four data types available in ESDL, namely `udisc`, `sdisc`, `cont` and `log`. They can be added to a class or module by selecting the corresponding element from the editor toolbar.

The ESDL method or process body itself does not contain variable declarations. Only if a variable is local to the current method/process can it be declared and initialized in the method body using a statement like the following:

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

5.2.5 Type Conversion

Whenever a basic arithmetic operator like `+`, `-`, `*`, `/` has operands of different types, the result is automatically converted to that of the strongest type used in the expression.

The order of types is (from weak to strong): `sdisc`, `udisc`, `cont`.

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

When assigning a value to a variable the data types must match. There is no explicit type casting. Only for the basic arithmetic types signed discrete, unsigned discrete and continuous does ESDL perform an implicit conversion.

```
cont tmp = 2;
```

A conversion of boolean and arithmetic types is *not* possible.

5.2.6 Primitive Methods

Every arithmetic type has a predefined interface which covers a set of basic math functions. The following messages are available for all arithmetic types:

Method	Receiver	Returns	Usage
<code>val.abs()</code>	arithmetic	arithmetic	absolute value of <code>val</code>
<code>val1.max(val2)</code>	arithmetic	arithmetic	the greater of two values
<code>val1.min(val2)</code>	arithmetic	arithmetic	the smaller of two values
<code>var.between(val1, val2)</code>	arithmetic	log	<code>var</code> between <code>val1</code> and <code>val2</code>

Tab. 5-1 Primitive methods for arithmetic types

The `var.between(val1, val2)` method corresponds to the `between:And:` element in block diagrams.

5.2.7 Literals and Constants

Literals are values like `12`, `6.1e4` or `true`. Every primitive type (boolean and arithmetic), can occur as a literal in an ESDL method. The data type of literals is implicit.

Constants are named values, such as `g = 9.81`. They are added to a class and declared in the same manner as variables. The Element Editor can be used to assign a value and flag a variable as a constant.

Some examples:

- `x = g.abs();`
The absolute value of the constant `g` is assigned to the variable `x`.
- `out1 = myvar.max(g);` or `out1 = g.max(myvar);`
The larger of the values `myvar` (a variable) and `g` is assigned to the variable `out1`.
- `out2 = myvar.min(.04);` or `out2 = (.04).min(myvar);`
The smaller of the values `myvar` and `0.04` is assigned to the variable `out2`.

5.2.8 Comments

A comment explains the purpose of a particular piece of ESDL code. There are two types of comments, commonly referred to as single- and multi-line comment.

Single-line comments are preceded by a double slash (`//`). The text that follows is ignored up to the end of the current line. Multi-line comments are delimited by `/*` and `*/`.

The comments used in an ESDL description are not transferred to the C code that is generated from that description.

5.2.9 Operators

In EDSL, method calls take precedence over all other operators. The order of precedence can be manipulated by adding parentheses to an expression.

Unary Operators:

The unary operators are `+`, `-` and `!` (not); the latter is used for boolean types. In addition, the increment and decrement operators `++` and `--` are available. They can be used as prefix or postfix operators.

Unary operators take precedence over all other operators. They associate right to left.

Arithmetic Operators:

The four arithmetic operators `+`, `-`, `*` and `/` can be used in ESDL. The modulus operator `%`, which calculates the remainder of an integer division, is also available.

The `*`, `/` and `%` operators take precedence over the binary `+` and `-` operators. Arithmetic operators associate left to right.

Comparison and Equality Operators:

The comparison operators are `>`, `>=`, `<` and `<=`. They are applied to arithmetic types and take precedence in this group.

The equality operators `==` and `!=`, which can be applied to both value and reference types, range next in the order of precedence.

Comparison and equality operators are binary. They associate left to right.

Logical Operators:

The logical operators `&&` and `||` (AND and OR) follow next in the order of precedence with the AND operator taking precedence over an OR.

Logical expressions are evaluated only until the truth or falsehood of the entire expression is determined. If, for example, the expression `a && b` is evaluated and `a` evaluates to `false`, it is redundant to evaluate the remainder of the expression. The evaluation of `b` has no impact on the result.

Logical operators are binary. They associate left to right.

Conditional Operator (MUX):

The conditional operator `?:` corresponds to the MUX operator in the block diagram editor. The operator has the general form $(a \ ? \ n \ : \ m)$ where a is a boolean, n and m must be of the same type. They can be any primitive type, boolean or arithmetic.

The value of a conditional expression depends on the value of a . If a is `true` in the above example, the value of the expression is n otherwise it is m .

The conditional operator is ternary. It ranges behind all binary operators in precedence. Association is from right to left.

Shorthand Assignment Operators:

In ESDL common shorthand assignments, such as `+=` or `*=` can be used. The $a \ += \ 4$ operation is a shorthand for the $a = a + 4$ assignment operation. Shorthand notation is available for the following operators:

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

Shorthand operators have lowest precedence. They associate from right to left.

Summary: Operator Precedence and Associativity:

The following table summarizes the precedence and associativity of operators in ESDL as described in the previous section.

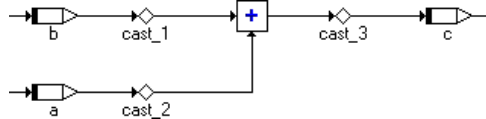
Operator	Associativity
<code>++ --</code>	right to left
<code>+ - (unary)</code>	right to left
<code>!</code>	right to left
<code>* / %</code>	left to right
<code>+ - (binary)</code>	left to right
<code>< <=</code>	left to right
<code>> >=</code>	left to right
<code>==</code>	left to right
<code>!=</code>	left to right
<code>&&</code>	left to right
<code> </code>	left to right
<code>?:</code>	right to left
<code>=</code>	right to left
<code>*= /= %= += -=</code>	right to left

Tab. 5-2 Operator precedence and associativity

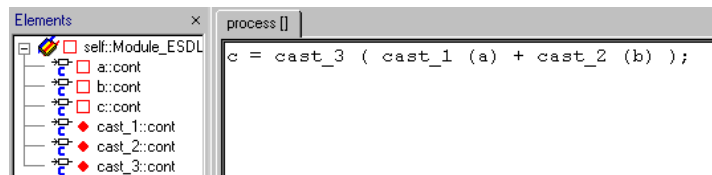
5.3 Implementation Casts in ESDL

Implementation casts (see chapter 4.2.4) are available in ESDL for modules and classes (except CT blocks).

In the specification of an operation in ESDL, implementation casts must be represented by their names. An addition with implementation casts that appears as follows in BDE:



is represented as a function in ESDL as such:



Here it is important that an implementation cast is written like a method call: it is always placed *before* the element to which it refers; the element is enclosed in parentheses, like a method argument. If the implementation cast is to be applied to the result of an operation, the *entire* operation must be enclosed in parentheses.

In the example above, `cast_1` refers to variable `a`, `cast_2` to `b` and `cast_3` to the result of the operation `a + b`.

If intermediate results of arithmetic operations are to be manipulated using an implementation cast, the corresponding intermediate results have to be enclosed in parentheses.

Thus, in this statement:

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

`cast_2` refers to the intermediate result of the operation,

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

while `cast_1` changes the overall result of the operation:

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

Note

An implementation cast in ESDL always refers to the value in the code that immediately follows the implementation cast.

It is important to note here, that the use of the syntax as described above is limited to implementation casts. The parentheses must contain an existing implementation cast; if you specify a standard type, such as `uint8 (a)`, an error message is displayed.

When using implementation casts, remember that they are not available for use with logical variables. If an implementation cast is applied to a logical variable, the code generator generates an error message.

5.4 Control Flow

The control flow elements can be used to determine the order of and conditions under which an ESDL function or statement is executed. The most common types are conditional structures and loops.

There are two types of conditional statements, `if...else` and `switch...case...default`, and two types of loop statements, `while` and `for`.

In addition, a `break` statement is available.

The control flow constructions in ESDL are described in more detail in the following subsections.

5.4.1 If...Else

The `if...else` statement can be used for simple conditional constructions. It has the general form

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

The `else` block can be omitted. When `expressionLog` is evaluated, the program decides whether to execute the `statementTrue` block. If not, the program either executes an existing `statementFalse` block or it continues without doing anything.

The `expressionLog` that controls the decision must be explicitly of type `log`. An arithmetic with a value of one or zero is *not* accepted.

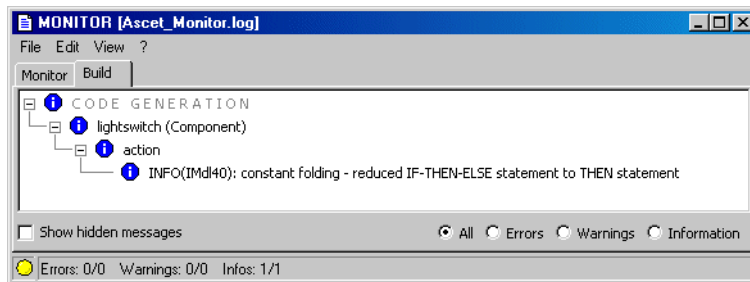
When the decision that the expression is always `true` can be made directly at the `if` statement, the construction is optimized in the generated C code. An example:

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

is reduced to:

```
cont=1;
```

When an optimization is performed, an information is given in the ASCET monitor window.



In the generated C code, however, *no hint* is given.

Note

*The decision whether optimization is performed is made locally at the `if` statement. If previous program parts would have to be considered to make the decision, **no** optimization takes place.*

5.4.2 Switch...Case...Default

The `switch...case...default` statement or, for short, the `switch` statement, can be used for more complex conditional constructions. It has the general form

```
switch (expressionsDisc) {
    case sDiscM: {
        statementM }
    ...
}
```

```

    case sDiscN: {
        statementN }
    default: {
        statementDefault }
}

```

The `switch` statement is a multi-way decision that tests whether the argument `expressionsDisc` matches one of the constant values `sDiscM` through `sDiscN` and branches accordingly.

Each case is labelled with a constant expression that must be of type `sdisc`. The corresponding block is executed if the `expressionsDisc` matches the value of the constant expression. The (optional) `case default` is executed if no other match can be found.

If the `default` case is not available and no match is found, the `switch` statement does nothing and control returns to the remainder of the software model.

The example below sets the value of a variable `scont` depending on the value of the `argSdisc`.

```

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

```

In this example, every block is terminated with a `break` statement. This causes the `switch` statement to be finished immediately after the block has been executed.

If the `case` blocks were not terminated explicitly, execution would continue immediately after a match has been found. In the above example, this means that for `sdisc=-1` the value of `scont` would first be set to 0 by the corresponding block and then set to -1 by the `default` block if the `break` statement was missing.

This phenomenon is commonly referred to as *fall through*. The remainder of a `switch` statement is always executed if a block is not terminated. Although this can be useful for multi-layered filtering it is generally regarded as poor style and should be avoided by terminating every `case` statement with a `break`.

5.4.3 While

The `while` loop is used to model a simple loop. It has the general form:

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

The loop condition `expressionLog` is evaluated. If it is `true`, the `loopStatement` block is executed and `expressionLog` is evaluated again. The loop exits when `expressionLog` evaluates to `false`.

In ESDL, the loop condition `expressionLog` must be of type `logical`.

5.4.4 For

The `for` loop stands out as one of the modelling features that are available in ESDL only. There is no equivalent in block diagrams.

The `for` loop has the general form

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

This is equivalent to

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

In the `for` loop, every component of the loop head, `initExpression`, `expressionLog`, and `incrExpression`, is optional. The loop condition `expressionLog` must be of type `logical`. It is set to `true` if omitted which results in an infinite loop.

Note

In ESDL, the components of the loop head must be simple expressions, comma-separated lists of expressions, such as `i=0, j=1` or `i++, j--`, are not accepted. In other words, it is not possible to use more than a single statement in either the `initExpression` or the `incrExpression`.

The following example is a simple combination of an `if...else` statement and a `for` loop:

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

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

The example writes values to an array. The `log` condition in the `if...else` statement determines which of the two loops is used to write values to the array.

Each of the loops iterates over the entire array and assigns a value to each cell. The value is either the result of `index * index` or the value of `index`.

5.4.5 Break

The `break` statement can be used to exit immediately from each of the control elements listed above and return to another enclosing statement or to the remainder of the model.

Since ESDL does not support labels in the model description, there is no labeled `break` statement that returns control to a label.

5.5 Methods

The functional description of a software model in ESDL is contained in methods. The methods perform calculations and manipulate data. They are invoked (or called) as operations on objects.

A method call has the general form

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

where `receiverClassName` is the name of the receiver object, which 'executes' the `doSomething` method. Parameters can be passed on as either a comma-separated list or a single parameter in the `parameterList`. Any expression can be a parameter, including method calls.

The following are valid method calls 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 is between valA and valB
```

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

Note

If a method has no parameters, the parentheses at the end of the method name still have to be supplied for the statement to be interpreted as a method call.

A method call can return a value, which can in turn be assigned to a variable in the method call. The variable must be of the same type as the return value.

```
aNumber = anArray.getAt(index);  
//assign value from index position  
anOffset = loader.resolve(true, 2.14);  
//assign value for 2.14, calculate using characteristic
```

If a method has a return value, the method body must be terminated with a return statement. The return statement can be followed by any expression that evaluates to the return type of the method.

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

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

A method call can return only a single value. If more than one value is to be passed on between modules or objects, an object can be used to hold these values (see section "Structures" on page 134).

Method calls cannot be nested in ESDL. The following statement is illegal:

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

It must be replaced with the following, legal statement:

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

Only if direct access methods are enabled for access to an object's variable, can a method call be nested. Hence, the following nested statement is legal if aa is a variable defined in anObject:

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

5.5.1 This

The pseudo-identifier `this` can be used in ESDL to call a method at the current component. If, for example, you want to call the private method `initCounter` at the current object, you can use the following statement:

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

If the `initCounter` method has a return value, you can assign it as follows:

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

The reference to the current object using the `this` identifier is optional in both these cases because it is implicit in the context. Hence, the above statements can be written as follows:

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

Only if the current object is to be passed on as a parameter to another method, is the reference using `this` needed.

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

Here, the identifier `this` passes on a reference to the current object.

Note

While ESDL accepts both the `self` and the `this` identifier, it is recommended to use `this` to ensure compatibility with Java syntax.

5.5.2 Access Control

In ESDL, both the methods and variables of a class can be declared as either public or private to control access to these elements and hide their implementation from other objects.

Private methods can be called and private variables manipulated only from within the current object. By contrast, public methods can be called and public variables accessed from both within and outside the current object.

Methods are declared public or private by assigning them to a corresponding diagram in the ESDL Editor. The default for new objects is to have a single public diagram `Main` which contains the `calc` method.

Users can create additional public methods in the same diagram or add a new diagram. Private methods must be created as part of a private diagram. The access rights to a method can be changed by moving it from one diagram to another.

An object `Caller` can access the public interface of another object `Receiver` if the latter has been imported by adding it to the `Elements` list for `Caller`.

New variables are created as private when they are added to the `Elements` list in the ESDL editor. They cannot be accessed from outside the current object. The status of a variable can be modified only in the element editor for that object (see ASCET user's guide, section "Editing Element Properties").

5.5.3 Direct Access Methods

Every public variable automatically adds two methods to the current object's interface, which are referred to as direct access methods. A direct access method can be called to access the data in a public variable. It can be used for both read and write access to that variable.

In the following example, suppose that the `VisibleObject` has two public variables named `free` and `all` respectively. Method calls from outside can be as follows:

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

Direct access methods are generated automatically and added to the public interface of an object whenever a variable is declared public. These methods do not have to be coded explicitly.

5.6 Composite Data Types

ESDL provides two groups of composite data types. The first group of composite types comprises common arrays and matrices, the second group, which contains one-dimensional tables, two-dimensional tables and distributions, is used for characteristic lines and fields.

Composite data types are explained in the following subsections, with arrays and matrices first and tables and distributions to follow suit.

5.6.1 Arrays

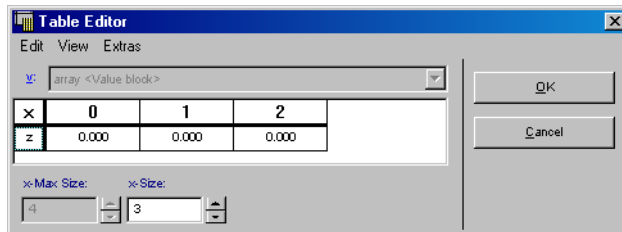
An array is a one-dimensional, indexed set of variables which have the same data type. In ESDL, arrays are available for all basic data types. The variables are accessed through the array index, the first index position is 0.

An array can be added to a module by adding it to the "Elements" list in the ESDL Editor. The array type can be specified in the element editor as any primitive type.

The array size and its data can be edited using the Table Editor dialog which is automatically opened when the data of an array are to be edited. You can specify both the current and maximum size of the array in the table editor.

The array size cannot be modified at runtime. The maximum size for arrays is 1024 elements.

The array data can either be edited in the table editor or filed in from a tab-delimited ASCII file (see ASCET user's guide, chapter "Editing Data", section "Array Editor").



In ESDL, elements of an array can be read and written to using the following syntax:

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

The first statement reads the value of the array element at position `index` and assigns it to the variable `val`, which must be the same type as the array. Since the array index count starts from 0, `myArray[3]` returns the fourth element of an array.

The second statement sets the value of the array element at position `index` to `val`, which must be the same data type as the array.

Public Interface:

Tab. 5-3 summarizes the public methods available of arrays.

Method	Returns	Usage
<code>length()</code>	<code>udisc</code>	get number of array elements
<code>getAt(index)</code>	type of array	get array element at position <code>index</code>
<code>setAt(val, index)</code>	<code>void</code>	set value of array element to <code>val</code>

Tab. 5-3 The public interface of arrays

5.6.2 Matrices

A matrix is a two-dimensional, indexed set of variables which have the same data type. In ESDL, matrices are available for all basic data types. The variables are accessed through the array indices `x` and `y`, the first index position is 0.

A matrix can be added and manipulated in the ESDL Editor in exactly the same manner as an array. The matrix size cannot be modified at runtime. The maximum size of matrices is 64 elements per dimension.

The elements of a matrix can be read and written to in ESDL using the following syntax:

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

The first statement assigns the value of the `matrix` element at position column `indX` and row `indY` to the variable `val`, which must be the same type as the matrix. Since the index count starts from 0, `myMatrix[2,3]` returns the third element in the fourth row of a matrix.

The second statement sets the value of the `matrix` element at position column `indX` and row `indY` to `val`, which must be the same data type as the matrix.

Public Interface:

Tab. 5-4 summarizes the public methods available for matrices.

Method	Returns	Usage
<code>xLength()</code>	<code>udisc</code>	get number of columns in matrix
<code>yLength()</code>	<code>udisc</code>	get number of rows in matrix
<code>getAt(indX, indY)</code>	type of matrix	get matrix element at position <code>indX, indY</code>
<code>setAt(val, indX, indY)</code>	<code>void</code>	set matrix element at position <code>indX, indY</code> to <code>val</code>

Tab. 5-4 The public interface of matrices

5.6.3 One-dimensional Tables

A one-dimensional table is used to model characteristic lines which describe parameter values in dependence of a given set of sample points rather than using an algorithm.

For each sample point x_n in the table, there exists a parameter value y_n which can be retrieved from the one-dimensional table. In addition, the table can cover the entire range of values between sample points using either linear or rounded interpolation.

An one-dimensional table can be added to a module by adding it to the Elements list in the ESDL Editor. The data type can be specified in the Element Editor as any arithmetic type.

The maximum size for one-dimensional tables is 1024 sample point : value pairs. Unlike arrays and matrices, tables are used as parameters in ASCET, that is, the sample points and values cannot be written to from within the model.

The table data can either be edited in the table editor or filed in from a tab-delimited ASCII file (see ASCET user's guide, chapter "Editing Data", section "The 1-D Table Editor").

The interpolation mode for sample points can also be specified in the table editor as rounded or linear. Rounded interpolation uses the value from the lower (left) sample point for a given point, whereas linear interpolation derives it from a straight line between sample values.

Public Interface:

In ESDL, tables can only be accessed using their public interface. Tab. 5-5 summarizes the public methods available for one-dimensional tables.

Method	Returns	Usage
<code>search(index)</code>	void	set the sample point of the table to index or calculate interpolation factor for index
<code>interpolate()</code>	type of table	get the value for the current sample point or interpolate it from the table
<code>getAt(index)</code>	type of table	set the sample point to index and get the corresponding value or calculate interpolation factor for index and interpolate the value

Tab. 5-5 The public interface of one-dimensional tables

Linear Interpolation:

The following example illustrates linear interpolation in one-dimensional tables. It uses a table `LLPR` that has the following values:

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

In general, the method `getAt(index)` is sufficient for the evaluation of characteristic lines. Linear interpolation for this example works as follows:

```

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 = LLrp.getAt(9000);
// calculates interpolation factor for 9000
// interpolates value for 9000 as 2.2 and
// assigns it to tmpVal

```

In some cases, though, separating the search and interpolate steps in tables can be more efficient, e.g. when generating code for experimental targets. In that case, linear interpolation is performed as follows:

```

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

```

5.6.4 Two-dimensional Tables

A two-dimensional table is used to model characteristic maps which describe parameter values in dependence of a given set of pairs of sample points rather than using an algorithm.

For each pair of sample points $(x_n : y_n)$ in the table, there exists a parameter value z_n which can be retrieved from the two-dimensional table. In addition, the table can cover the entire range of values between sample points using either linear or rounded interpolation.

A two-dimensional table can be added and manipulated in the ESDL Editor in the same manner as a one-dimensional table. The maximum size for two-dimensional tables is 64 pairs of sample points and corresponding values.

Public Interface:

In ESDL, tables can only be accessed using their public interface. Tab. 5-6 summarizes the public methods available for two-dimensional tables.

Method	Returns	Usage
<code>search(indX, indY)</code>	void	set the sample points of the table to <code>indX</code> and <code>indY</code> or calculate interpolation factor for <code>indX</code> and <code>indY</code>
<code>interpolate()</code>	type of table	get the value for the current sample point or interpolate it from the table
<code>getAt(indX, indY)</code>	type of table	set the sample point to <code>indX</code> and <code>indY</code> and get the corresponding value or calculate interpolation factor for <code>indX</code> and <code>indY</code> and interpolate the value from the table

Tab. 5-6 The public interface of two-dimensional tables

Linear Interpolation:

The following example illustrates linear interpolation in two-dimensional tables. It uses a table `LLpr1` that has the following values:

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

As with characteristic lines, the method `getAt(indX, indY)` contains everything that is needed for the evaluation of characteristic maps. Linear interpolation for this example works as follows:

```
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
```

With characteristic maps, too, separating the search and interpolate steps in tables can be more efficient. In that case, linear interpolation is performed as follows:

```
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 = LLrp2.interpolate();  
// interpolates value for (4,4) as 3.143 and  
// assigns it to tmpVal
```

5.6.5 Distributions and Group Tables

Characteristic lines and fields can be related to each other by using the same set of sample points. In ASCET, such a shared set of sample point is modelled as a distribution, tables that use the sample points in a distribution are referred to as group tables.

A distribution is an array of sample points. The sequence must be strictly monotone increasing. Distributions can be used for both types of tables (one- and two-dimensional). Two-dimensional tables require a distribution for each dimension.

Using distributions and group tables can significantly reduce the time and memory required for computations since interpolation factors are computed only once and can be reused over a set of tables.

Adding a group table in the ESDL Editor consists of first adding a distribution and then a group table. When the group table is added, the system prompts for the corresponding distribution. Since the ESDL Editor cannot be used to reassign distributions to existing group tables, distributions should always be created before adding the tables.

The Table Editor can be used to edit the data of both distributions and tables. It does not accept distributions which violate strict monotony. Data can also be filed in from tab-delimited ASCII files.

Public Interface:

Unlike plain tables, group tables do not have a `getAt` method. Instead, the public interface is "split" between the distribution, which has a `search` method, and the group table, which has an `interpolate` method.

A two-dimensional table requires sample points to be set for both distributions before the corresponding value can be interpolated.

Tab. 5-7 shows the public interface of distributions in ESDL.

Method	Returns	Usage
<code>search(index)</code>	void	set the sample points of the distribution to index or calculate the interpolation factor for index

Tab. 5-7 The public interface of distributions

Tab. 5-8 shows the public interface of group tables in ESDL.

Method	Returns	Usage
<code>interpolate()</code>	type of table	get the value for the current sample point or interpolate it from the table

Tab. 5-8 The public interface of group tables

5.7 Structures

In ESDL, structures (or records) are modelled using classes. A class can be used as a complex container element which holds any number of variables. If a variable in a class is public, it can be read and written to from ESDL using direct access methods.

Classes that are used as container elements are accessed in the same manner as other classes in ESDL. The first step is always to add the class to the Elements list of the ESDL Editor to make it available in the context of the current class. Variables can be declared public in the Layout Editor for the parent object.

The variables can then be accessed from within ESDL using the simple direct access method syntax:

```
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
```

For group tables and distributions, this procedure does *not* work.

In ESDL, classes can be nested to model self-referential structures.

Note

A complex assignment such as `VisibleObject.anArray(myArray)` is not legal in ESDL, it does not assign the values in the `myArray` parameter to the `anArray` element. Complex statements can, however, be used to pass on a reference to another object.

5.8 Messages

In ASCET, an additional concept of messages as real-time language constructs is used for interprocess communication. Messages, in this sense, are used as protected global variables in the real-time environment.

Messages are available only in modules. From within a module, a message is merely a variable that can be read, written to or both. Whenever a process runs, the operating system creates copies of all its messages. These copies are accessible only to that instance of the process that created them.

Hence, if the same message is used by various processes, each process gets its own copy of the message. This strategy is used by the real-time operating system to ensure data consistency over multiple processes.

Messages are fully supported in ESDL, they can be used in all modules. A message is added like all other elements in the Elements list by selecting the corresponding icon from the ESDL Editor toolbar. Messages can be added as

- send messages—the current module can write to this variable,
- receive messages—the current module can read this variable, or
- send and receive messages—the current module can read and write to this variable.

In ESDL, messages are accessed through assignment statements:

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

Public Interface:

Tab. 5-9 summarizes the public methods available for messages.

Method	Returns	Usage
<code>receive()</code>	void	read message
<code>send()</code>	void	write message

Tab. 5-9 The public interface of messages

5.9 Resources

Similar to messages, resources are available only in modules. As described in section 3.1.3, they have two access methods, `reserve` and `release`. In ESDL, these methods can be used as shown in the following example:

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

Tab. 5-10 summarizes the public methods available for messages.

Method	Returns	Usage
<code>reserve()</code>	void	reserve a resource
<code>release()</code>	void	release a resource

Tab. 5-10 The public interface of resources

5.10 Mathematical Functions

ASCET comes with a comprehensive library of pre-defined elements. They can be used as building block for new modules and classes.

For model descriptions in ESDL, additional mathematical functions are provided in the system library. The mathematical functions are defined in the class `Etas_Systemlib_CT\Classes\MathFcn` and can be accessed after this class has been added to the Elements list of the ESDL Editor.

The following examples show how to access mathematical functions from an ESDL model description.

```
// 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); }  
}
```


The following table summarizes the functions available in the `MathFcn` class. The return and parameter types are the same for all mathematical functions, they accept variables of type continuous as parameters, the return type is continuous, too.

Method	Operation
<code>pi()</code>	returns 3.141592654
<code>sin(x)</code>	sine of x
<code>cos(x)</code>	cosine of x
<code>tan(x)</code>	tangent of x
<code>asin(x)</code>	$\sin^{-1}(x)$ (arc sine)
<code>acos(x)</code>	$\cos^{-1}(x)$ (arc cosine)
<code>atan(x)</code>	$\tan^{-1}(x)$ (arc tangent)
<code>sinh(x)</code>	hyperbolic sine of x
<code>cosh(x)</code>	hyperbolic cosine of x
<code>tanh(x)</code>	hyperbolic tangent of x
<code>sch(x)</code>	hyperbolic secant of x
<code>csch(x)</code>	hyperbolic cosecant of x
<code>coth(x)</code>	hyperbolic cotangent of x
<code>exp(x)</code>	exponential function e^x
<code>log(x)</code>	natural logarithm $\log_e(x)$, $x > 0$
<code>log10(x)</code>	base 10 logarithm $\log_{10}(x)$, $x > 0$
<code>pow(x, y)</code>	x^y
<code>sqrt(x)</code>	square root of x
<code>abs(x)</code>	absolute value $ x $
<code>sign(x)</code>	sign function returns: -1 if $x < 0$; 0 if $x = 0$; 1 if $x > 0$
<code>limit(m, x, n)</code>	limiter returns: m if $x \leq m$; x if $m < x < n$; n if $x \geq n$
<code>max(x, y)</code>	returns the greater value of x and y
<code>min(x, y)</code>	returns the smaller value of x and y
<code>fmod(x, y)</code>	floating point remainder of x/y, same sign as x
<code>ceil(x)</code>	returns smallest integer value not smaller than x
<code>floor(x)</code>	returns largest integer value not larger than x

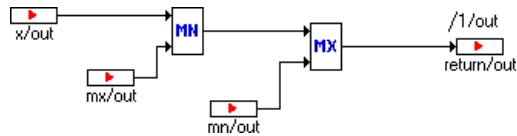
Tab. 5-11 Mathematical functions in ESDL

5.11 Accessing Block Diagrams from ESDL

This section guides you through building a simple limited integrator in ESDL. The integrator uses a limiter element from the `SystemLib_ETAS` folder to determine the bandwidth of the outgoing signal.

The limiter element has a single method `out` with three parameters `mn`, `x`, `mx`. The `out` method either returns `mn` if $x < mn$, `x` if $mn \leq x \leq mx$, or `mx` if $x > mx$.

The block diagram for the limiter element is displayed below.



To build the integrator element

- In the Component Manager, create a new ESDL module and rename it to `Integrator-Limit`.
- Open an ESDL Editor for `Integrator-Limit`.
- In the "Elements" list, add a continuous variable named `mem`. The integrator's memory stores the value of the outgoing signal.
- In the "Elements" list add the limiter module from the following folder:
`Systemlib_ETAS\Nonlinears\Limiter`.
- In the Methods list, add the methods `out`, `reset` and `compute`.
You can either rename the default method `calc` to `compute` or delete it.

- Use the Interface Editor to edit the corresponding method interfaces as follows:

Method	Arguments	Returns
compute	cont mx cont in cont mn	void
out	void	cont
reset	cont initVal	void

- Enter the ESDL code for each method and save the method. The ESDL code for each method is listed below.

```

reset(initVal)
mem = initVal;

cont out()
return mem;

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

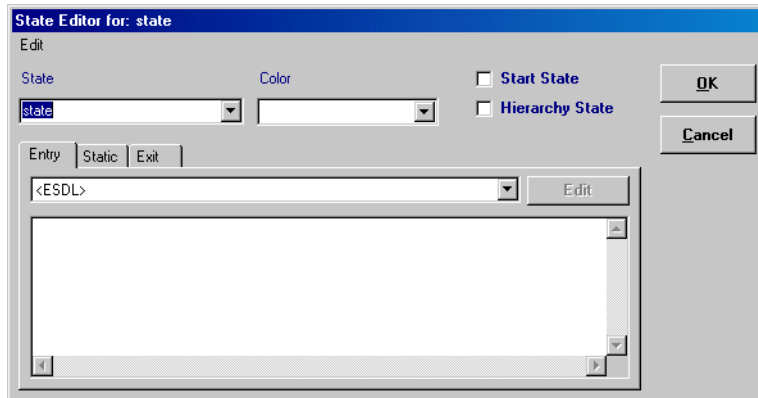
```

The example shows how to use an existing module as a building block for a new one. The second statement in the `compute` method limits the integrator signal. The limiter's `out` method returns the signal value or the lower or upper bound, which is assigned to the integrator's memory.

5.12 Using ESDL in State Machines

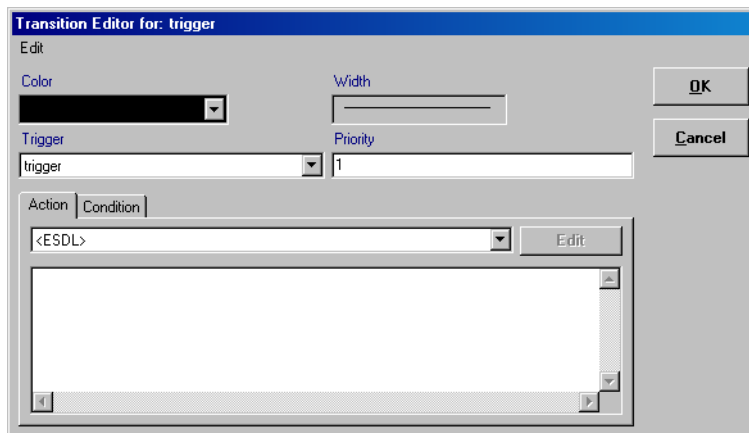
When modelling state machines in ASCET, the description in ESDL is often more compact than block diagrams. ESDL can be used to describe both states and transitions between states.

Typically, a state can have three up to three different actions, which are labelled entry, static and exit. They are performed when the state is entered, while it is active, and when the state is terminated.



The actions in a state can be edited in the State Editor. They can be specified in ESDL if the <ESDL> option for the corresponding action is selected. This activates the text field for the action which is a simple ESDL editor. From this editor the output and input variables of the state machine and all other items in the Elements list of the state machine can be accessed.

A transition between states usually has a condition that triggers the transition to another state; it can have an action as well, which is executed when the transition is performed.



The transitions between states can be edited in the Transition Editor. Again, conditions and actions can be specified in the text field in ESDL after the <ESDL> option has been activated and all items in the elements list can be accessed.

In all text fields of both editors, standard ESDL code is used as in the examples above. The one important point to remember in ESDL syntax is that the expression entered in the "Condition" tab returns a boolean and is not terminated by a semicolon. You find more about editing actions and conditions in ESDL in the ASCET user's guide, chapter 4.2, section "Conditions and Actions in the State Diagram".

5.13 Overview: ESDL Features Compared

ESDL vs. Block Diagrams

The following table presents an overview of differences in model descriptions using ESDL and block diagrams.

	ESDL	Block Diagrams
<code>this</code>	x	o
<code>self</code>	x	x
<code>%</code> operator	x	o
<code>++</code> , <code>--</code> operator	x	o
<code>for</code> statement	x	o
atomic sequences	o	x

Tab. 5-12 Synopsis: ESDL vs. block diagrams

Reference: ESDL vs. ANSI C

The following table presents an overview of the main differences between the ESDL modelling language and the ANSI C programming language.

	ESDL	ANSI C
bit data type, shift operations	o	x
string data type, string operations	o	x
<code>continue</code> statement	o	x
pointer arithmetic	o	x
preprocessing	o	x

Tab. 5-13 Synopsis: ESDL vs. ANSI C

Reference: ESDL vs. Java

The following table presents an overview of the main differences between the ESDL modelling language and the Java programming language.

	ESDL	Java
inheritance	o	x
dynamic instantiation	o	x
polymorphism	o	x
method overloading	o	x
explicit type casting	o	x
error handling	o	x
garbage collection	o	x

Tab. 5-14 Synopsis: ESDL vs. Java

6 Body Specification with Block Diagrams

With the block-oriented description language of ASCET embedded control systems can be specified graphically. It is the graphical equivalent of the ESDL language used for specifying control systems textually.

This section describes how to specify software modes using block diagrams in ASCET. The following section starts with a brief introduction to the graphical description of components, which is followed by an overview of the graphic modelling language.

The overview section presents linguistic means available in block diagrams:

- Elements
- Expressions
- Statements

Block diagrams and ESDL are, for the most part, functionally equivalent in ASCET. The differences between block diagrams and ESDL are summarized in section "ESDL vs. Block Diagrams" on page 141.

6.1 Graphical Description of Elements

Every element and operator used in a component is graphically represented by a diagram item in the form of a rectangle. The interaction between these elements is represented by lines connecting the corresponding diagram items.

The interface of an element is represented graphically by pins (Fig. 6-1). Each argument of a method is represented by an argument pin (with a little arrow head pointing towards the block) at the block frame. The return values are represented by a return pin. The call to a method is associated with its return pin. Methods without arguments or return values are represented by a method pin.

- ↳ Method Pin
- Argument Pin
- Return Pin

Fig. 6-1 The representation of pins in graphical blocks

The name of the element is placed underneath the rectangle. An icon can be used to illustrate the functionality of an element. The position of the pins can be changed by the user

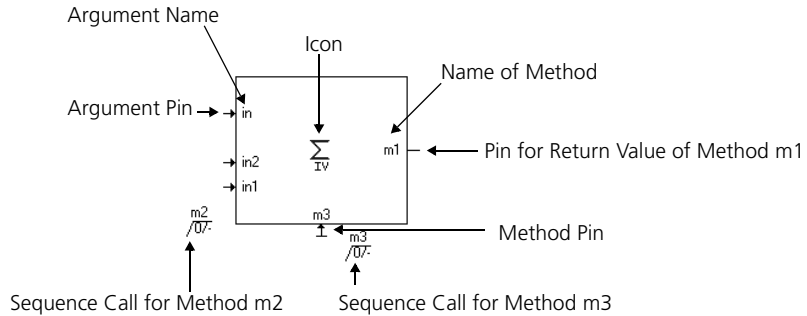


Fig. 6-2 The graphical block for a complex element.

The example in Fig. 6-2 shows a complex element with three methods. Method m_1 has one argument and a return value. Method m_2 has no return value and is represented by its arguments, method m_3 has neither arguments nor a return value and is represented by a method pin.

6.1.1 Basic Elements

Elements are represented as rectangular blocks with the arguments and return values represented as pins. Each element has a name that is placed underneath the block by default, but this position can be changed.

All basic types have a fixed interface and their graphical representation is also fixed.

Basic Scalar Elements

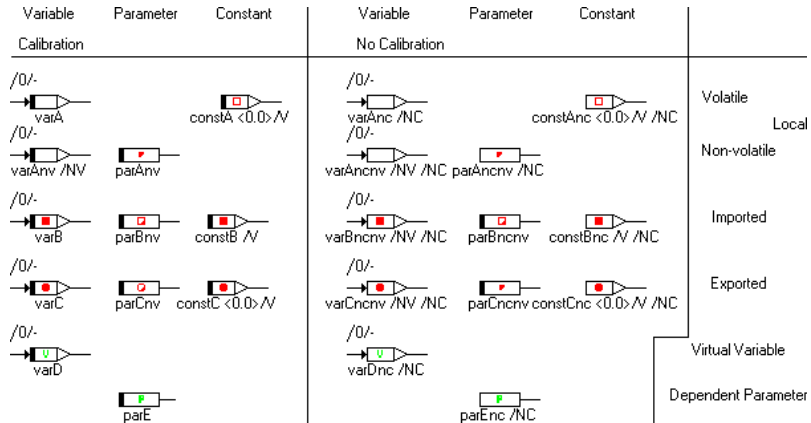
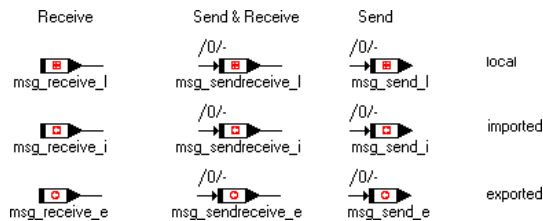


Fig. 6-3 The graphical representation of basic elements

Basic scalar elements have one argument pin for setting a new value (if their value can be set), and one return pin for reading the current value. The icon inside the block represents the kind of the element. The scope of an element is also indicated by the icon: a solid red square represents an imported element, a solid circle an exported one. If the kind of the basic scalar element does not permit writing to it (e.g. parameters) the corresponding pin is missing. Elements that can be calibrated have a small black box on the left side.

Messages

Messages are the input and output variables of processes. Depending on the message type, they are displayed with one or two pins. The figure shows messages with the attributes *Calibration* and *volatile*; changing one or both attributes makes the display change as shown for variables in Fig. 6-3.



Literals

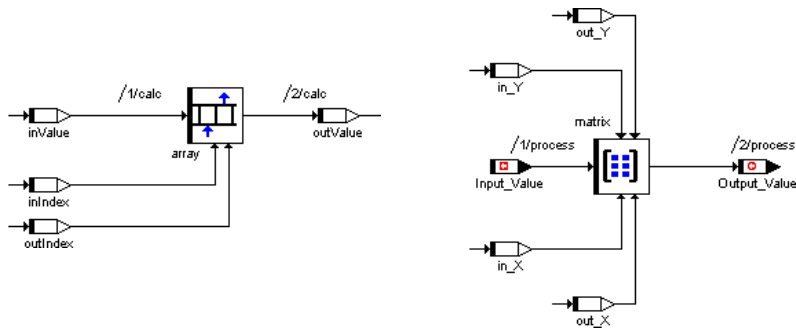
Literals are represented by small blocks, with the value of the literal inside the block:



Arrays and Matrices



An array or matrix has two methods, one for setting the content of a specific element and one for retrieving it. The read and write operations can occur independently of each other. The value to be written to the array is represented by the left (argument) pin, the corresponding index by the bottom left pin. The result of reading from the array is represented by the return pin and the index by the bottom right argument pin.



Matrices are represented similarly, but each method takes two index arguments. The x-index is represented by the bottom left pin, like the index of an array. The y-index is represented by the pin at the top of the block with the top left pin being the index for writing to the matrix, and the top right pin the index when reading from it.

When arrays or matrices are to be passed as method arguments, or returned as return values, this is done with the help of *Get and Set ports*. These are made available via the **Get/Set Ports** pop-up menu function in the drawing area.

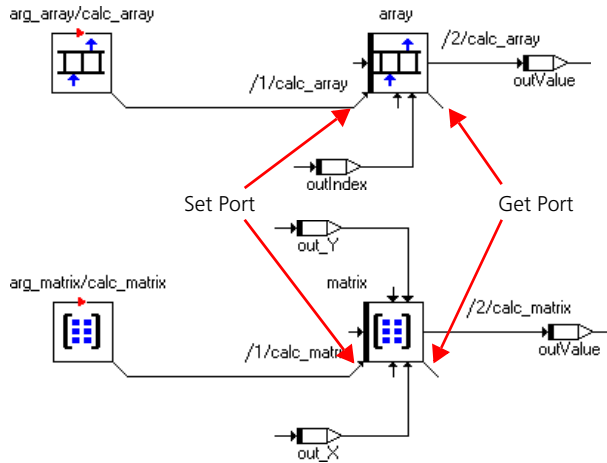


Fig. 6-4 Get and Set ports for arrays and matrices

The Get port provides a pointer to the entire data content of the respective element; by the Set port directs the element to access a certain memory area.

An example: In Fig. 6-4, `array` reads from the memory area used by `arg_array`, while `matrix` reads from the memory area used by `arg_matrix`. The pointers to the respective memory areas are passed via the Get and Set ports. It is important that writing to the Set port is performed as the first step of the method; otherwise, inconsistencies arise.

Note

The same mechanism is used to pass classes, too.

Characteristic Tables

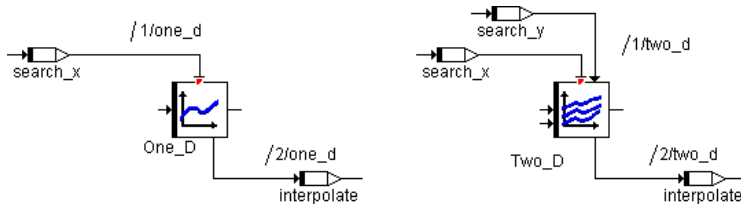


Depending on the dimension, characteristic tables, including fixed tables, have one or two argument pins on the left side where the sample values are supplied, and one return pin where the value of the interpolation is given.

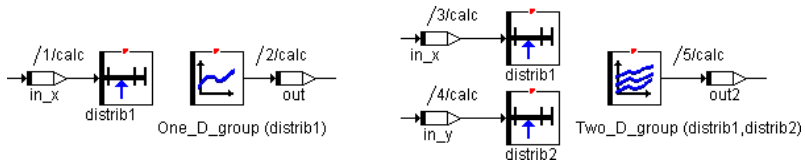


The above representation corresponds to using the `getAt` method in ESDL (cf. the respective sections on page 129 and page 131).

As with ESDL, the search and interpolate steps in tables can be separated in the block diagram editor. To do so, the extended table interface has to be made available via the **Extended Interface** pop-up menu function in the drawing area.



A distribution has one argument pin for the sample value on the left side of the distribution. A group table has one return pin on the right side. It contains no own sample point distribution, but references one or two distributions instead. Group tables and distributions do not have an extended interface.



As with arrays and matrices, Get and Set ports can be made available via the [Get/Set Ports](#) pop-up menu function.

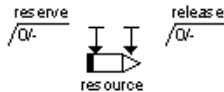
Note

If you want to pass characteristic tables as method arguments, you have to embed them in classes, and pass the class via the Get port.

Resources



Resources are represented by a block with the two methods `reserve` and `release` at the top. Both methods have no arguments or return values and are represented as method pins:



Implementation Casts



Implementation casts are represented by a small diamond with two pins.



6.1.2 Elements of User-defined Type

The methods, arguments and return values of elements of user-defined type are represented by argument or return pins at the graphical block. The user can define the layout of the representation for each user defined type. Get and Set ports can be made available for these elements, too.

6.2 Expressions

Expressions are formed in block diagrams by connecting elements or other expressions with operators. Like in ESDL, expressions are built up recursively, as follows:

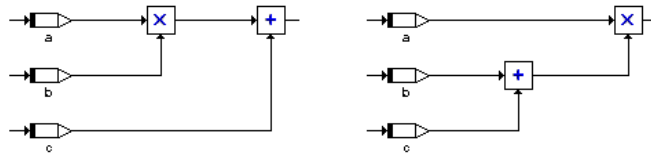
- An element is an expression.
- The result of an operator is an expression (the operands itself are expressions).

- The return value of a method call is an expression. If arguments are supplied to the method, these arguments also belong to the expression.

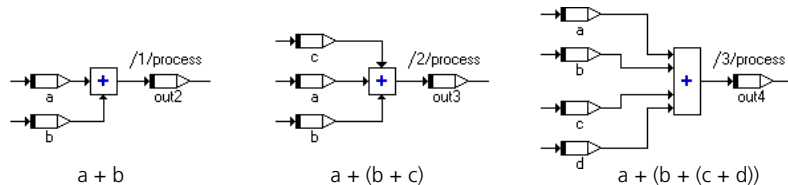
The range of an expression is therefore limited by the base expressions in that expression, which are either elements or return values of methods without arguments.

Expressions are built graphically by connecting the return pins of elements or operators with the argument pins of methods or other operators.

There are no precedence rules for operators in the BDE, since the expressions are “bracketed” by the way the lines and operators are connected. The following example shows the difference between the expressions $(a*b)+c$ and $a*(b+c)$ in the graphical representation.



The evaluation order of the arguments of operators is sometimes very important. In the graphical representation this sequence is always from top to bottom, except for the four basic arithmetic operators with at most three inputs. The order of evaluation is illustrated in the following diagram:



In block diagrams the number of arguments to the operators is often limited to a maximum of 10 or 20 inputs. The evaluation order of method arguments depends on the order in which they are defined. Since the layout of an element can be changed, the order in the layout must not coincide with that in the definition.

6.2.1 Arithmetic Operators



The meaning of the operators is the same as in ESDL. The following operators are available: Addition, Subtraction, Multiplication, Division, Modulo. The addition and multiplication operators can have between 2 and 10 arguments. The subtraction and division operators have only two arguments.

6.2.2 Comparison Operators

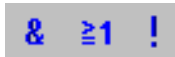


The comparison operators are identical to their counterparts in the textual representation with ESDL. The following comparison operators are available:

- Greater Than
- Less Than
- Less or Equal
- Greater or Equal
- Equal
- Not Equal

The Equal and Not Equal operators can also be applied to non-arithmetic elements.

6.2.3 Logical Operators



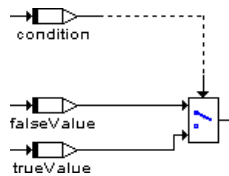
The meaning of the logical operators And, Or and Not is identical to their meaning in ESDL. The And and Or operators can be applied to more than two operands.

6.2.4 Conditional Operators

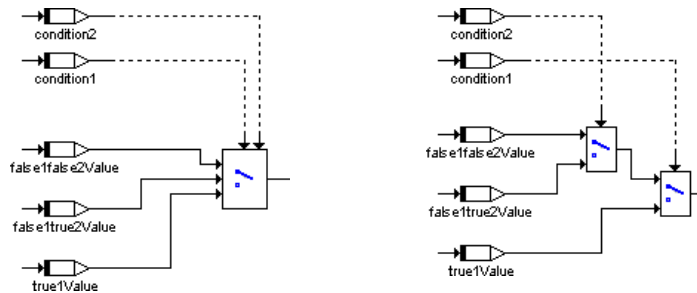
Multiplex Operator



The conditional operator (`? :`) is named Multiplex operator (for short: Mux) in the graphical representation. The graphical representation of (`condition ? trueValue : falseValue`) is as follows:



The multiplex operator can also be used directly with several arguments (left image); the right image shows the identical functionality built as a cascade of several Mux operators:

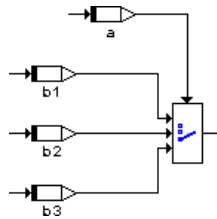


The above example is equivalent to `(condition1 ? (true1Value : condition2 ? (false1true2Value : false1false2Value)))`, i.e., the first argument has priority over the others. A cascaded Mux operator with n logical condition arguments can select between $n+1$ arguments between which it switches. The type of the arguments is arbitrary, but all arguments must be of a compatible type.

Case Operator



The case operator is a special case of the conditional operator. It does not take a logical value but a switch value of type unsigned discrete. The Case operator selects one of the arguments depending on the switch value. If the switch value is 1, the first argument is selected, if it is 2 the second is returned and so on. If the switch value falls outside the range, the last argument is selected.



The above example is equivalent to $((a=1) ? b1 : ((a=2) ? b2 : b3))$.

6.2.5 Other Operators

Besides the operators described so far, the following operators are also available:

- Max and Min
- Between
- Abs
- Negation

Max and Min Operators

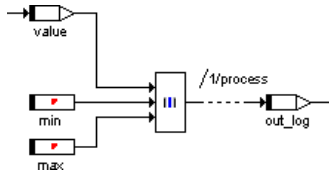
MX MN

The Max and Min operators return the maximum or minimum of the arguments. Both operators can have 2 to 20 arguments; they can be applied only to arithmetic elements.

Between Operator



The Between operator checks if the argument `value` lies between the limiters `min` and `max`. If this is the case, the logical return value `out_log` is `true`, otherwise it is set to `false`.



The graphical representation is equivalent to `out_log = ((value >= min) && (value <= max))`. The argument and both limiters have to be either `cont` or `discrete`.

Abs Operator



This operator returns the absolute value of the argument. Argument and return value have to be both either `cont` or `discrete`.

Negation Operator

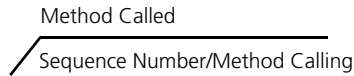


The Negation operator returns the negative value of the argument. Argument and return value can be `cont` or `discrete`; if the argument is `cont`, the type of the return value should be the same.

6.3 Statements

Graphical specifications of components can be hierarchically distributed over several diagrams. In a diagram one or more methods or processes can be described which can be executed independently of each other. The order in which calculations are executed, as well as the particular method or process a calculation belongs to is determined by sequence calls.

For each statement of a block diagram, there is a sequence call that assigns it to a process or method. The order within a process or method is determined by the sequence number that is part of the sequence call. A sequence call is represented graphically as follows:



With the sequence numbers the order of the operations belonging to one method or process can be determined by the user. A built-in sequencing algorithm can be used to assign sequence numbers that correspond to the evaluation order of standard block diagrams.

A sequence call generally consists of three fields:

- The name of the method called.
- The name of the method or process calling.
- The sequence number determining the position of the called method in the calling method or process.

In the case of scalar elements, the name of the method called is left blank as this is always the assignment of a new value.

There are three kinds of statements:

- Assignment statements
- Method calls
- Control Flow Statements, e.g. `if...then...else`, `while`

6.3.1 Assignment

An assignment statement is the assignment of the value of an expression to an element. In case of an assignment to a complex element, only an element of the same type can be assigned. The assignment is then not the assignment of a value but of a reference.

A special case is that of assigning a value to the return value of a method. The associated sequence call must be the last sequence call of that method.

6.3.2 The Break Statement



The return from a method or process can also be established by the break statement. This statement does not have to be the last of a method or process.

6.3.3 Method Call

An assignment is a special case of a method call. When calling a method in a block diagram, the corresponding sequence call has to be filled in properly and the arguments to the method have to be supplied.

6.3.4 Control Flow

The following control flow statements are available in block diagrams:

- If...Then
- If...Then...Else
- Switch
- While

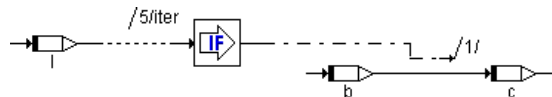
All control flow statements evaluate a logical expression and, depending on the result, activate a control flow branch which may contain several statements. The statements represented by sequence calls are connected to the control flow by connectors.

The sequence number of the sequence call determines the order of the statements connected to the activated control flow branch.

If...Then



The If...Then statement evaluates a logical expression and activates a control flow branch if the result is `TRUE`. The control flow output is connected to one or more sequence calls which are triggered whenever the control flow branch is activated. Whenever the input expression evaluates to `TRUE`, the connected sequence calls are executed.



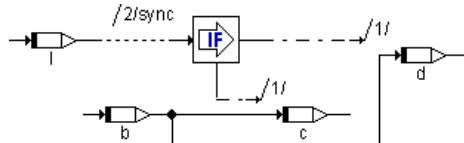
The example above is equivalent to

```
if (1) {  
    c = b  
};
```

If...Then...Else



If...Then...Else is similar to If...Then, but has two control flow branches. Depending on the value of the logical expression, the left or right branch is executed, the right branch is executed if the value is `True`, the left one if it is `False`.



The example above is equivalent to

```
if (1) {  
    d = b;  
}  
else {  
    c = b;  
};
```

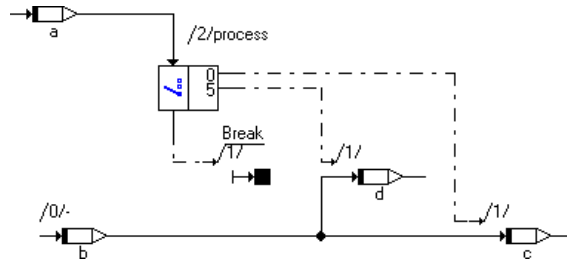
As for the `if...else` statement in ESDL, the generated code is optimized when the expression for `If...Then` or `If...Then...Else` is always `true`. Section "If...Else" on page 120 describes how the optimization works.

Switch



The `Switch` construct is similar to the `Case Operator`. A `Switch` evaluates a `signed discrete` or `unsigned discrete` value and, depending on that value, activates different control flow branches. These branches are separated from each other, so that a "fall through" like in the `switch` construct in C is not possible.

For each alternative the value for the branch can be defined by the user. The last branch at the bottom is the default branch that is executed if the input value does not equal any of the values at the branches.



The example above is equivalent to

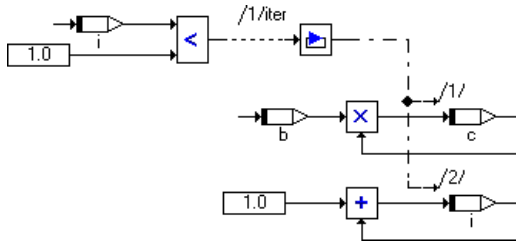
```
switch (a) {
    case 0: c = b; break;
    case 5: d = b; break;
    default: break;
};
```

While



The only loop construct available in block diagrams is the `while` loop. Care has to be taken to avoid infinite loops or loops unsuitable for real-time applications.

Similarly to the `If...Then` statement, the control flow is activated when the value of the logical expression is `True`. The operation is executed as long as the value of the logical input remains `True`. Therefore, the value of the logical expression should be manipulated in the while loop. In order to avoid infinite loops, the number of maximal loop iterations can be limited to a fixed, user definable number.



The example above is equivalent to

```

while (i<10) {
    c = b * c;
    i = 1 + i;
};

```

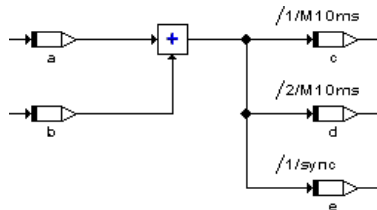
6.4 The Semantics of Block Diagrams

Each part of a block diagram is assigned to a process or method. The execution order is determined by the sequence numbers in the sequence calls. When a process or method is activated, all statements whose sequence calls are attached to that process or method are executed in the order given by the sequence numbers.

In contrast to standard block diagrams, an operation is executed only on demand, i.e. when its sequence call is activated. The order of execution is similar to the left-to-right principle of standard block diagrams: before an operation, for example an addition, can be performed, the values for all its arguments have to be computed.

The order of evaluation of the arguments of methods of user-defined components is given by the order of their declaration. This order, however, may not coincide with the order implied by the diagram, as the argument pins can be arranged arbitrarily at the block frame.

The evaluation of operands etc. is directly associated with the statements that use the results. This may result in multiple evaluations of an expression.



In this example, the addition is executed three times, for each of the assignments to the variables c , d , and e . The addition is used in assignments in two different processes. Without multiple execution, it would not be clear in which of the processes the addition should be executed. The expression $a + b$ is evaluated twice in the process `10ms`.

6.4.1 Graphical Hierarchies

In order to structure a graphical specification, graphical hierarchies can be used. Graphical hierarchies do not influence the semantics of a block diagram but are used for structuring only. A hierarchy contains a part of the block diagram. The lines that cross the border of the hierarchy, i.e. that connect elements inside the hierarchy with those outside, are represented by pins. In ASCET 5.2, an icon can be assigned to hierarchies in the block diagram editor.

7 Body Specification in C

The specification of the body of methods and processes can be implemented in C code as well as in the form of block diagrams and ESDL. As with the other specification methods, only the body of a method or process has to be specified. The method declaration, the function head and frame, and the data instantiation and initialization are generated automatically.

In contrast to specifications in either ESDL or with block diagrams, components in C code are specified on the implementation level, rather than on the model level.

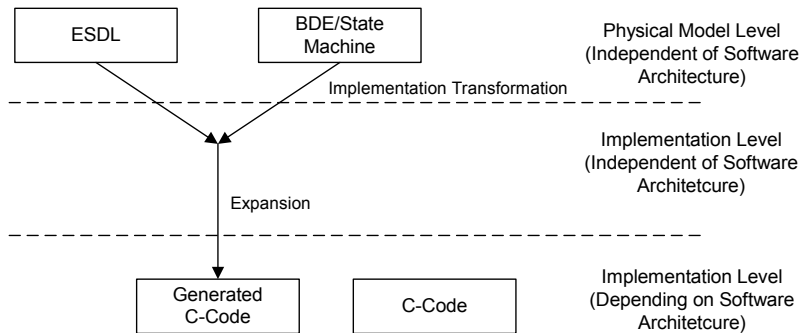


Fig. 7-1 From physical model to implementation

This has several important consequences:

- There is no transformation from the model to the implementation level.
- For each code variant (different target, different specification level, different implementations) the C code can be different. As a consequence, the code must be specified separately for each variant.
- The C code has to be adapted to the software architecture of the code generated by ASCET when user-defined types are used. This is because the interface is generated and the exact naming convention for the generated C functions depends on the expander and may not be transparent to the user. In the present expander, the identity tag of the class is used in the name for the generated functions in order to guarantee a unique name space.

7.1 Structure

A component described with C code has the same structure as if it was described with ESDL or as a block diagram. The C code describes the body of methods or processes. Each code variant is stored separately.

The specification of a component in C code depends on:

- The target, e.g. whether the C code is for the PC, PPC or for a specific controller CPU. Here the code can vary, since, for instance, a controller CPU has special registers that have to be addressed directly, or the endian format is different.
- The specification level. The C code can be intended to represent the physical level. In that case the implementation level coincides with the physical level as far as possible, e.g. the type `continuous` is represented as a 64-bit float. Alternatively the C code can be on the implementation level of fixed point arithmetic.
- The chosen implementation, if the C code is on the implementation level, since the C code depends on the implementations of the variables, particularly on their quantizations.

7.1.1 Methods and Processes

For each method or process a C function is generated. The function head is generated automatically, the C code is only used in the function body itself.

Example:

The body of the method `calc()`

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

could result in the following generated code (including function head), depending on the software architecture required for the experimental target:

```
void QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_calc (struct  
QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_Obj *self) {  
...  
    /* BEGIN handwritten code */  
    /* calc 1 */a = b + d;  
    /* calc 2 */c = a * c;  
    /* END handwritten code */  
...  
}
```

The names of the functions generated for the methods and processes of components depend on the code expander and the software architecture of the generated code. The user has no influence on these names. Depending on the code expander a unique name space is achieved, i.e. methods at different classes can have the same name without any naming conflicts. In the above

example the identity tag for the component is used to generate the unique name `QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_calc` for the method `calc`.

7.1.2 Variables and Function Parameters

The variables of a component are held in a data structure that, like the function heads, is automatically generated. The user has no influence on this data structure. A part of this data structure consists of the instance variables of the component, which can be used in any method. Therefore they have to be passed to all generated functions. This data structure also depends on the code expander and the exact naming is therefore hidden from the user.

In the above example, the component has a data structure of its own that is passed to the generated function for the method `calc`. The data structure could look like this:

```
struct QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_Obj {
    ASDObjectHeader objectHeader;
    real64_Obj *a;
    real64_Obj *b;
    real64_Obj *c;
    real64_Obj *d;
};
```

The element names must be valid ANSI C identifiers. In addition to the reserved keywords of C, the names `self` and `this` are reserved.

Note

When specifying components in C code, the user must ensure that the names of functions called in the method body do not collide with the names of variables defined in the interface of that same component.

Accessing Elements:

To allow easy access to the elements of the component, a macro is defined automatically for each element. Each element can then be accessed simply by its element name.

The public elements defined in other components can be accessed from within C functions using the notation `DefiningObject.PublicElement`. Access is restricted to basic elements, arrays and matrices. The public interface of complex elements defined in other components, e.g. using the `getAt`, `setAt` or `search` and `intepolate` methods as in ESDL, cannot be accessed from C functions.

Automatically Generated `define` Statements for Instance Variables:

```
#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
```

Working with Basic Elements :

For basic types, the method names of these types can be used, as explained in chapter 3 "Types and Elements" on page 89. When accessing arrays or matrices, the index operator '[']' can be used in a C-like manner.

Since the method names of a user-defined type depend on the expander, the method of user-defined types can only be called with the knowledge of the exact generated function name for that method. In the above example the function name `QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_calc` is generated for the method `calc`.

When using elements defined in ASCET, these elements are of a model type (either basic or user-defined). Basic types have the following default implementation, which is taken on the physical level:

- `continuous = real64`
- `udisc = unsigned int32`
- `sdisc = signed int32`
- `log = int16.`

Note

Elements of type `logical` should not be used as numbers in the C code, since this depends on the default implementation, which is subject to change in further releases of ASCET.

The default implementation is replaced by the user-defined implementation when switching the specification level (e.g. fixed point code). Elements of model type `logical` can be represented for instance as a bit, and can therefore not be used as a number in the C code.

Messages:

Messages are part of an intra-task (intra-process) communication concept used within ASCET models (see chapter 1.3). To achieve data consistency, the ASCET code generation has to create additional message copies.

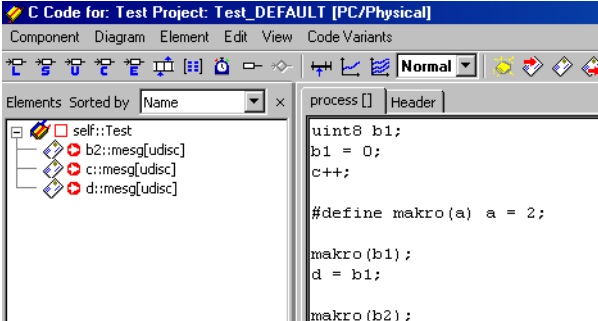
If messages are used within the functional code (read/write access), additional code is required to ensure safe copying of the current values from message originals to the local copies. Within the process body, only these local copies are used. At the end, all local copies which could change their value within the process body must be written back to the message originals.

In ESDL and block diagram components, ASCET generally detects very well, which messages are changed within a process. However, this functionality is of limited availability when using C code for the body specification. Here, the user has to take care of data consistency on his own.

In general, ASCET is not able to detect where and when a variable is written in user-specified C code. ASCET recognizes only a few special cases where, e.g., the variable name is followed by a `=`, or where assignment operators like `++` are used. If a variable is changed within a macro, an extern function, or via adress operators and pointer arithmetic, ASCET does *not* detect the change.

When messages are used, this behavior results in message copies being created at the beginning of the process, but—under certain circumstances—not written back at the end.

A simple example shall illustrate this. The module shown below contains the messages `b2`, `c`, and `d`. The messages `c` and `d` are directly written, `b2` is used within a macro.



The screenshot shows a window titled "C Code for: Test Project: Test_DEFAULT [PC/Physical]". The window contains a menu bar (Component, Diagram, Element, Edit, View, Code Variants) and a toolbar. Below the toolbar is a tree view of elements sorted by name, showing a hierarchy: self::Test, b2::msg[udisc], c::msg[udisc], and d::msg[udisc]. The main editor area shows the following C code:

```
process [] Header
uint8 b1;
b1 = 0;
c++;

#define makro(a) a = 2;

makro(b1);
d = b1;
makro(b2);
```

In the generated code, copies are initially created for all three messages (1). However, since only `c` and `d` are accessed in a way ASCET can recognize, only these two message copies are written back at the end (2). The change of message `b2` that occurs in the macro, is not recognized and gets lost.

```

/*
-----
*   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; } (1)
    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; } (2)
    RESUME_INTERRUPTS
}

```

Arguments:

Arguments of methods are mapped to function parameters in the parameter list of the function generated for the method. These are also accessed by the name of the argument.

Local variables:

Following the general C rules, function local variables can be declared in the method body. Here only variables of a C data type can be declared, not however of an ASCET model type. In particular, no local variables of a user-defined type can be used in components within body specifications in C.

```

real64 i;

for (i=0; i < 10; i++)

```

```

{
    sum = sum + a[i];
}

```

Since there is a code variant for each implementation variant, the user can define the local variables and their data types with respect to the implementation variant.

Characteristic lines and maps:

Characteristic lines and maps defined in the component are evaluated via three subroutines each (as in ESDL).

Table LLpr from section "One-dimensional Tables" on page 129 is again used as an example for a characteristic line.

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

The CharTable1_getAt_real64_real64(charline, index) subroutine is usually sufficient for the evaluation of characteristic lines. Linear interpolation for this example works as follows:

```

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

```

In some special cases, though, separating the search and interpolate steps in tables can be more efficient. In these cases, the subroutines CharTable1_search_real64(charline, index) and CharTable1_interpol_real64_real64(charline) are used.

```

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

```

Table LLpr2 from section "Two-dimensional Tables" on page 131 is again used as an example for a characteristic map:

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

The CharTable2_search_real64_real64(charline, indX, indY) subroutine is usually sufficient for the evaluation of characteristic lines. Linear interpolation for this example works as follows:

```

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

```

In some special cases, though, separating the search and interpolate steps in tables can be more efficient. In these cases, the subroutines CharTable2_search_real64_real64(charmap, indX, indY) and CharTable2_interpol_real64_real64_real64(charmap) are used.

```

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

Besides the description of the methods in the form of C code, a header can be defined for macros and for included files. This header has a local range restricted to the component. Therefore, no extra header file is generated, but the definitions are copied into the generated C code file.

7.2 External Source Code

Existing C code can be integrated by importing external C code source files. For this purpose, one C code file with a corresponding header file can be attached to each code variant of a component. The C code file contains standard C function definitions, the header file contains the corresponding function declarations and structure definitions. The defined functions can be called through the standard C conventions. It is possible to pass pointers and share defined structures between methods or processes of the component and the functions in the attached C code.

As an alternative to using a C code file, an object file with a corresponding header file can be attached to a component. Like the header files of the component itself, the range of the header files of the attached sources is local, i.e. they are copied into the generated C code.

The attached C file is compiled separately and linked to the other (generated and compiled) C source files. As a consequence, this compiled unit exists only once within any given context. If the code and the included data is shared between multiple instances of the same component, all instances share the same compiled unit.

Note

The data in an attached C file is shared between multiple instances of the component, and not instantiated for each of the instances.

Additionally it is possible to have `include` statements in the C code. The include files however are not stored in the database, but are stored on the file system. The `include` statement must contain the file path to these `include`

files. The C code therefore depends not only on items in the database but also on the file structure of the current installation. Therefore care has to be taken when exchanging data, since these files are not known to the ASCET system.

Note

When using include files the user must take care of the correct references to these files on their own.

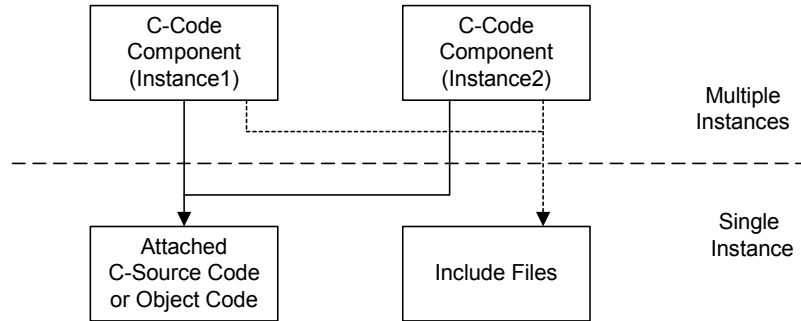


Fig. 7-2 Using external source code

7.3 Programming Model Interface

In earlier ASCET versions, the names of classes and methods could only be used in C code, if they were labeled with an escape symbol ("@"). By means of this mechanism, the so called Programming Model Interface (PMI) was activated. In ASCET 4.x and 5.x, class and method names are recognized automatically, i.e. no escape symbol is necessary and the PMI is used by default (see description of code generation options in the ASCET manual for details). Therefore, the escape symbol is obsolete and should not be used any longer when modelling in C code. For backward compatibility, however, the escape symbol can still be used when modifying the respective code generation option.

7.4 Access Macros

Similar to access methods in ESDL, ASCET offers access macros for the usage with C code components. For the following operations, special macros are defined. By means of these macros, the user can apply pre-defined operations in the C code. The macros are described in the following sections.

Direct Access

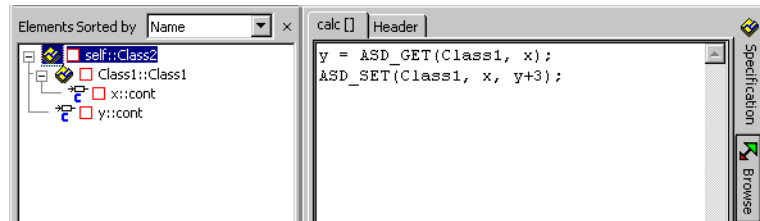
Elements of classes embedded into C code components can be accessed directly using the macros

```
ASD_GET(receiver, variable);
```

and

```
ASD_SET(receiver, variable, value);
```

Example:



Length of Arrays

The current length of an array can be determined using the macro

```
ASD_LENGTH (receiver);
```

Example:

```
z = ASD_LENGTH(array);
```

Resource Access

Resources can be reserved and released using the macros

```
ASD_RESERVE(resource);
```

and

```
ASD_RELEASE(resource);
```

Access to Private Methods

Private methods can be accessed using

```
self.
```

Example:

```
y = self.method_private(x);
```

Making Arrays Available for Usage in External C-Code

Using the macro

```
ASD_USE_ARRAY_EXTERNAL(array)
```

array access can be converted from the ASCET internal representation to the standard C code representation. The macro is a synonym for:

```
&array[0]
```

Example:

```
y = c_function(ASD_USE_ARRAY_EXTERNAL(array));
```

8 Continuous Time Systems

The comprehensive capabilities of ASCET are utilized to model discrete systems for the functional development of controller software and for the simulation of control units. In contrast, the control system associated with the control unit represents a continuous time physical system that is described by differential equations.

Examples for continuous time systems are the drive train or the wheel suspension of vehicles (mechanical system), the combustion process in the cylinder chamber (thermodynamic system), the brake circuit of a vehicle (hydraulic or pneumatic system), and the vehicle battery (electric or electrochemical system). In addition, there are increasingly also mechatronic systems in which, e.g., the mechanics of an actuator is connected with a local electronic control, or an intelligent sensor processes the physical signal electronically.

ASCET supports the model design and simulation of such continuous time systems by means of so-called CT blocks. CT stands for *Continuous Time* and refers to elements that are modeled or calculated in quasi-continuous time increments. The continuous time modeling in ASCET is based on state space representation, the standard description form in the design of continuous time systems. This representation allows the description of CT basic blocks by nonlinear ordinary first-order differential equations and nonlinear output equations. ASCET provides several real-time integration methods to solve these differential equations efficiently.

The continuous time model can be structured in modular and hierarchical blocks. Continuous time models can be combined by ASCET controller specifications to create combined models, so-called hybrid projects. These hybrid projects can be used to test a controller specification against a model of the actual technical processes that need to be controlled.

The model and the simulation experiment are strictly separate; a model contains the modular and hierarchical system description while an experiment contains the selected data set, the integration algorithm, and the selected visualization configuration including an input method for parameters. The results are accurate, reusable models and high flexibility. At the experiment level, each model variable can be flexibly altered and measured. The chosen integration step size and the integration algorithm can be changed during the simulation, without any time-consuming recompilation of the model or the current experiment.

8.1 Structure of Continuous Time Models

The following sections describe the various structuring options for a model with basic blocks, structure blocks, and graphical hierarchies.

8.1.1 Modeling with Basic Blocks and Structure Blocks

Models of continuous time systems can be structured in modules and hierarchies. The fundamental element is the continuous time basic block, or CT basic block, in which the partial model is described in the form of differential equations, algebraic equations, formulas and assignments using the high-level languages ESDL or C.¹

Continuous time blocks (CT blocks) consist of inputs, outputs, parameters, and discrete and continuous states with several dimensions, scopes and data types. In addition, continuous time and discrete equations and output equations as well as an initialization and termination sequence are also supported. State events, software and hardware events (interrupts) can also be handled.

More complex continuous time models can be assembled to CT structure blocks using the Block Diagram Editor (BDE). Using the Block Diagram Editor, several CT basic blocks and/or CT structure blocks can be assembled and combined. Fig. 8-1 shows a simple CT structure block composed of two CT basic blocks.

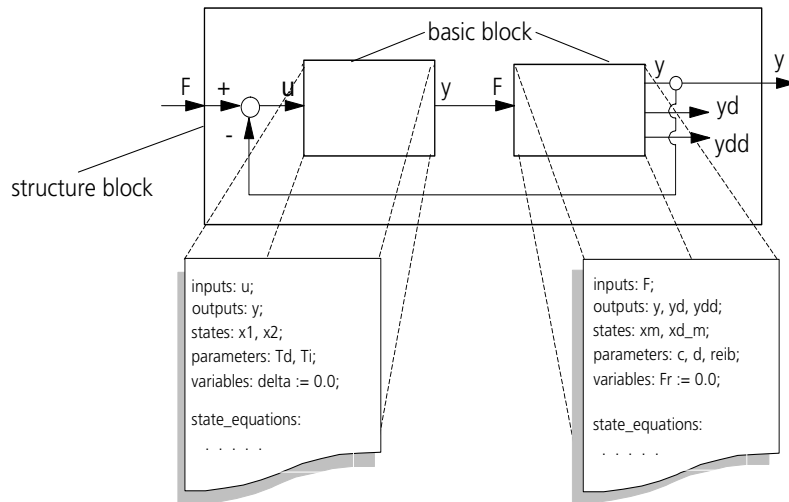


Fig. 8-1 CT structure block composed of two CT basic blocks

¹: C should be used in imperative, exceptional cases only because ASCET provides automatic verification functions (semantic checks, computing sequence) only for ESDL.

Several CT structure blocks and CT basic blocks can in turn be combined to one new CT structure block. Fig. 8-2 shows the possible structuring options with CT structures.

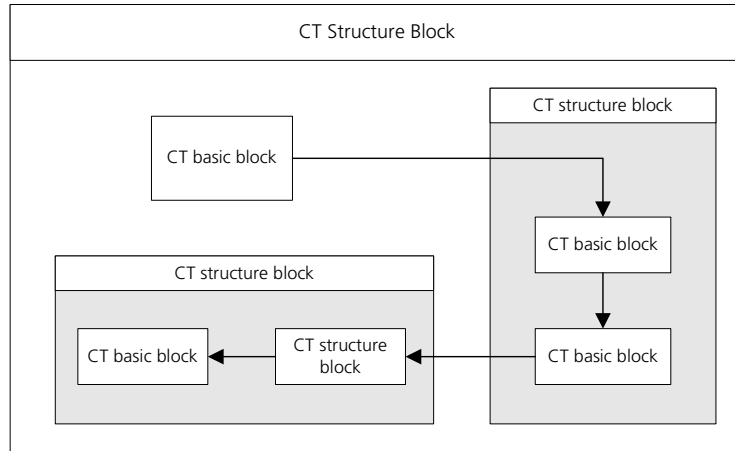


Fig. 8-2 Modeling with CT structure blocks

The correct computing sequence of the CT blocks is determined automatically. CT basic blocks and/or CT structure blocks can be combined together with Standard ASCET structures to build hybrid projects.

CT basic blocks are used to describe small physical components such as brakes, wheels, etc. CT structure blocks serve to describe more complex entities such as a power train facility or a complete vehicle model. CT basic blocks and CT structure blocks are each stored in the database and are available for other models. In this way, it is possible to easily build a model library. Modifications to blocks or structures are automatically distributed to all models within one database. This has the advantage that basic elements have to be maintained at one place only while corrections are automatically adopted by all models included in the same database. On the other hand, it must of course be ensured that the basic elements remain compatible.

8.1.2 Modeling with Graphical Hierarchies

A CT structure block composed of many CT basic blocks and/or CT structure blocks can be designed more clearly by combining several related CT blocks in a graphical hierarchy (refer to Fig. 8-3). Graphical hierarchies and CT structures can be combined into new hierarchies-the processing sequence is not affected by these graphical hierarchies. In the Block Diagram Editor, graphical hierarchies are indicated by a double-line frame.

Graphical hierarchies are especially used when the individual CT blocks have strong cohesion and require a fixed computing sequence within an integration step. By using graphical hierarchies, algebraic loops (refer to section "Algebraic Loops" on page 201 and section "Difference Between Graphical Hierarchies and CT Structure Blocks" on page 204) that may be caused by CT structure blocks can be avoided. The correct computing sequence is ensured by automatic sequencing. Graphical hierarchies cannot be stored individually but only together with the corresponding structure block.

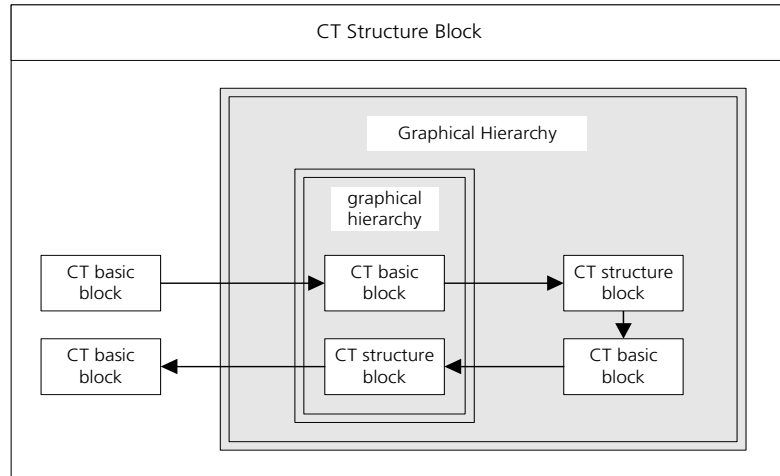


Fig. 8-3 Graphical hierarchy

8.1.3 Experiments

Basic and structure blocks can be evaluated in a simulation experiment. In the experiment, the integration method, the model stimulation, and the visualization of results are selected and specified. Several experiment settings can be stored for a (partial) model.

8.1.4 Projects and Hybrid Projects

The real-time experiment is defined in a project. Both basic blocks and structure blocks can be used in a project. Furthermore, it is only in the project where individual integration methods and their step size can be assigned to each integrated basic block or structure block. This allows allocating more CPU time to the model part with high dynamics than to other, less dynamic model parts, if the processor capacity is limited.

For a model in which the controller and control system models are to be combined, a hybrid project can be defined, i.e., a project that contains both CT blocks and standard ASCET components. A hybrid project thus allows the simulation of the control system and the control unit in one model (hybrid simulation).

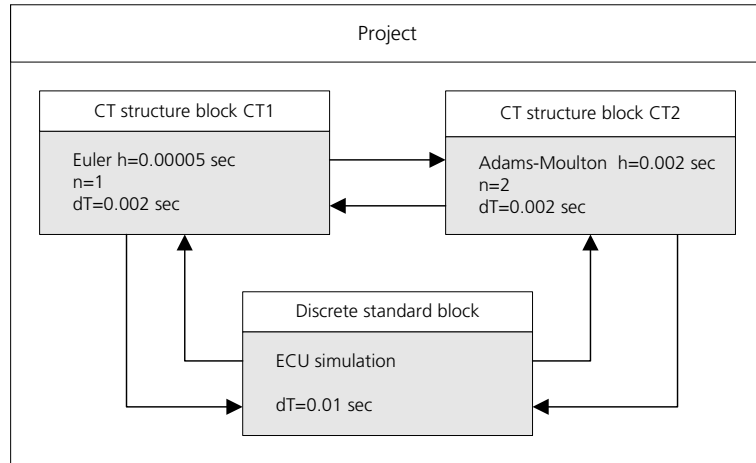


Fig. 8-4 Project

The communication between individual CT blocks and individual controller modules is performed by explicitly connecting inputs and outputs in the Block Diagram Editor (for details on projects, refer to chapter "Projects" on page 13).

The experiment can be conducted on-line on the real-time simulation hardware or off-line on the PC (unless special hardware has to be available or integrated for the experiment).

8.2 Solving Differential Equations – Integration Algorithms

Due to the complexity of the equations in continuous time models and frequent non-linearities, it is generally not possible to solve them by analytic methods. It is therefore necessary to solve the differential equation system using a numeric integration algorithm.

If only CT blocks are simulated in a CT structure, ASCET uses a global integration algorithm. The combination with discrete controller models is possible at the project level only (combined modeling in a hybrid project). Projects also support modeling with several CT structures using different integration methods.

To ensure high flexibility and short iteration cycles for modeling and simulations, the configuration of the integration method, i.e., the actual integration method and its integration step size, can be selected and modified interactively during the experiment.

There is no ideal integration method that fits all types of models. The speed and accuracy of the individual algorithms varies for different model characteristics such as non-linearities, discontinuities, and dynamic behavior. A general statement regarding the speed of each method cannot be given, as the step size is adapted to suit the model and integration method best. However, some guidelines for selecting a suitable integration method are given below. Detailed information can be found in the literature, e.g.,

Addison, C. A.; Enright, W. H.; *et al.*, A Decision Tree for the Numerical Solution of Initial Value Ordinary Differential Equations. *ACM Transactions on Mathematical Software* 17, 1, March 1991, Chapter "Continuous Time Integration Algorithms".

ASCET provides the following integration methods:

- Euler
- Mulstep 2
- Heun
- Adams-Moulton 2
- Runge-Kutta 4

To solve more complex or stiff differential equations that need more precise calculation, ASCET provides the following variable-step iterative integration methods:

- Dormand/Prince RK5
- Calvo 6(5)
- Dormand/Prince RK8
- Implicit RK2
- Implicit RK4
- Implicit Gear 1
- Implicit Gear 2

During calculation, these integration methods adapt the step width used iteratively in order to achieve a certain given precision. Therefore, they *cannot* be used for real-time calculations.

Due to technical reasons, the implicit integration methods can only be used with newer Borland and Microsoft compilers. They cannot be used with the Borland C 4.5 compiler shipped with ASCET. These methods are taken from the GNU Scientific Library. The integration method Gear 4 provided in previous versions of ASCET is not available anymore.

8.2.1 Integration Methods – Overview

It is assumed that the differential equation exists in its state form:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}, t); \text{ with } \mathbf{x}(t=0) = \mathbf{x}_0$$

The table below lists some characteristics of the implemented integration methods:

- The global error order p of the discretion error that is proportional with h^p , where h is the integration step size.
- The number of function evaluations per integration step. Each time, the local variables are reset and the `nondirectOutputs`, `directOutputs`, `derivatives` methods are executed. This, combined with the integration step size, can be used to estimate the speed of the method.
- Single-step/multi-step methods (SSM/MSM): Single-step methods only use the last estimated value for the next step, whereas multi-step methods take the last n estimates into account.
- A predictor-corrector method (P-C) first uses an integration method to calculate an estimate which is then corrected using a second method.
- Fixed or variable step size.

The table below contains a summary of these characteristics for integration methods with fixed step size (for MSM, the time when the function is computed or when the break points are taken into account is indicated in parentheses).

Integration Method	Error Order	Function Evaluations/ Step	SSM/MSM	P-K	Step Size
Euler	1	1 (t)	SSM	no	fixed
Mulstep 2	2	1 (t)	MSM (t-h, t)	no	fixed
Heun	2	2 (t, t+h)	SSM	yes	fixed
Adams-Moulton	2	2 (t, t+h)	MSM(t-h, t)	yes	fixed
Runge-Kutta 4	4	4 (t, t+h/2, t+h/2, t+h)	SSM	no	fixed

To ensure that the integration methods can be applied in real-time, each method is implemented using relatively few function evaluations per integration step and a correspondingly low error order.

Euler

The Euler integration method is the simplest integration method available. A single-step method with only one function evaluation per integration step, its cycle time is the smallest, making it relatively fast and especially suitable for real-time simulations.

Mathematical Formula

$$x(t+h) = x(t) + h * f(x, t)$$

Its stability range is high, however, at the trade-off of a higher discretion error that, at the same step size, is typically higher than with the other methods (lowest order).

Mulstep

The Mulstep integration method is a multi-step method which is used for models without heavily varying eigenvalues. The cycle time of one integration step is only slightly higher than for the Euler method since only one function evaluation is performed per integration step. However, the error order is 2.

Mathematical Formula

$$x(t+h) = x(t) + h(3/2 * f(x, t) - 1/2 * f(x, t-h))$$

Heun

The Heun integration method is used for models without heavily varying eigenvalues. The cycle time is twice as long as with the Euler method.

Mathematical Formula

Predictor: $x(t+h) = x(t) + h * f(x, t)$ (Euler)

Corrector: $x(t+h) = x(t) + h/2 * (f(x, t) + f(x, t+h))$

Adams-Moulton

The Adams-Moulton integration method is also suitable for models without heavily varying eigenvalues. In contrast to the previous algorithms, the model should exhibit a smooth behavior. The cycle times for the Adams-Moulton and Heun algorithms are almost the same.

Mathematical Formula

Predictor: $x(t+h) = x(t) + h/2 (3f(x, t) - f(x, t-h))$ (Adams-Bashforth)

Corrector: $x(t+h) = x(t) + h/2 (f(x, t) + f(x, t+h))$

Runge-Kutta 4

The Runge-Kutta integration method is best suited for models without heavily varying eigenvalues. This integration method is very robust for this type of model. It is the slowest, but also the most accurate method at comparable step sizes. It is therefore possible to increase the step size considerably.

Mathematical Formula

$x(t+h) = x(t) + h/6 (K_1 + 2K_2 + 2K_3 + K_4)$

where

$K_1 = f(x, t)$

$K_2 = f(x + K_1 * h/2, t + h/2)$

$K_3 = f(x + K_2 * h/2, t + h/2)$

$K_4 = f(x + K_3 * h, t + h)$

Integration Methods With Variable Step Width

For experiments that need very precise calculation, the step width usually has to be reduced. This can increase the time used for calculation significantly. Models employing stiff differential equations often render the calculation using fixed-step integration methods infeasible. Adaptive integration methods are controlled by a target error margin. The step width is only reduced for those parts of the model where it is needed. Because the step width (and therefore the time needed for calculation) varies, these integration methods are not real-time capable.

If the desired precision cannot be reached due to the parameter settings, the experiment issues a warning in the ASCET monitor window. This happens when the maximum iteration depth is set too low or the minimal step width is set too high.

9 Continuous Time Basic Blocks

Continuous time basic blocks (CT basic blocks) are generally used to describe small, independent physical components that can be used in various model scenarios. Basic blocks can be specified using the CT block editor. The block interface is specified interactively and the dynamics of the physical component are described by differential and algebraic equations.

9.1 Basics

Continuous time basic blocks are specified either in C code editor or in the ESDL editor. The two editors are slightly different for the specification of CT blocks. The internals of the blocks, i.e., the differential and algebraic equations as well as the control structures, are described within pre-defined methods. The proper computing sequence required for correct, continuous time modeling is derived automatically (sequencing). The pre-defined method structure cannot be modified by the user.

Basic blocks are used to describe models by means of nonlinear ordinary first-order differential equations (ODE) and nonlinear output equations. To describe a system of higher order, it has to be converted into several differential equations of first order. The table below illustrates the transformation of a second-order system into its representation in the state space.

One 2nd-order differential equation	Two 1st-order differential equations
$T^2 \cdot x'' + 2.0 \cdot d \cdot T \cdot x' + x = K \cdot \text{in};$	$x' = xp;$
	$xp' = (K \cdot \text{in} - (2.0 \cdot d \cdot T \cdot xp) - x) / T^2;$

Tab. 9-1 Resolution of a second-order differential equation

The equations can be written in ESDL or C. The use of ESDL ensures a target-independent specification and advanced semantic checks. When using C, the entire functionality of the C programming language is available. The drawback of C is that it is not possible to perform a semantic analysis. The use of ANSI C enables largely target-independent modeling, however, this is not the case if special language dialects such as for special hardware optimization is used. Furthermore in C, the block's behavior has to be specified as direct or nondirect.

9.2 Available Elements and Methods

Continuous time basic blocks differ in some elements from discrete modules or classes. The following elements exist:

- Inputs

- Outputs
- Continuous state
- Discrete state
- Steplocal variables
- Parameters
- Dependent parameters
- Constants
- OneD / TwoD table parameters

Each element type can have different dimensions, scopes, and data types (refer to section "Block Interfaces" on page 185). The figure shows the various data types (and their associated icon) and the available methods for working with these data.

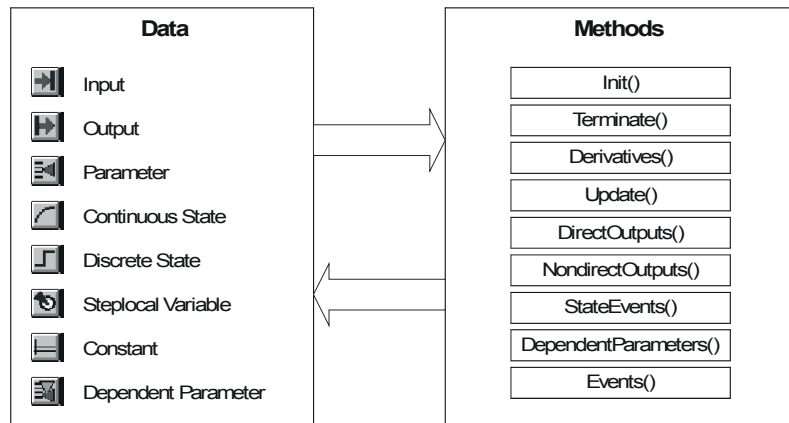


Fig. 9-1 Different data types and methods

9.2.1 Modeling With Continuous Time Basic Blocks

Within a continuous time basic block, the internals of the system to be modeled can be described using the ESDL model description language or directly in C. The target-independent ESDL modeling language provides advanced semantic verification ensuring a correct model. Modeling directly in C, therefore, should be confined to target-dependent real-time blocks only. In general, the use of ESDL is recommended.

The behavior of the block is described within a fixed framework, i.e., with a fixed number of methods. Each method has a specific purpose, e.g., the calculation of derivations or outputs. In contrast to standard ASCET models, the execution sequence is fixed (refer to section "Computing Sequence" on page 187), and the methods are scheduled automatically.

9.3 Block Interfaces

The elements (interfaces, storage elements) for modeling continuous time basic blocks are slightly different from those used for discrete modules or classes. The following describes the available element types.

Inputs: Block inputs have to be described using *Inputs*. At each evaluation step, all input variables are read.

Outputs: Block outputs have to be described using *Outputs*. At each evaluation step, all output variables are updated.

Continuous State: The description of ordinary differential equations requires state variables. Each state variable functions as a "storage element"; an example is the distance and velocity of a moving mass point. Continuous state variables are only used by the differential operator $\text{d}t$.

Discrete State: A discrete state variable is a storage element. It can be used to keep a variable value from one calculation step to the next, e.g., the value of a counter. Discrete state variables are equivalent to the variables in discrete classes or modules. Discrete state variables cannot be used by the differential operator $\text{d}t$.

Steplocal Variables: Steplocal variables are used to store intermediate values during the calculation of an evaluation step. These variables are visible in all block methods. The value of a steplocal variable is valid only in one evaluation cycle; the variable is reinitialized at the beginning of each iteration step. If the value must be evaluated in a different method, the execution sequence of the methods has to be considered (ensure writing before reading).

Parameters: Parameters are used to create a physical model. Normally, a parameter corresponds to a characteristic property of a real system, such as mass, length, or attenuation constant. A generic model library can be systematically built by means of efficient parameterization. Parameters can be varied during the simulation in the experiment environment (using the Calibration Editor).

Dependent Parameters: If one parameter depends on another parameter, e.g., parameters described in different coordinate systems, it should be recalculated only if the other, affecting, parameter has changed. This type of

parameter behavior can be described by dependent parameters. They are calculated only in case of changes asynchronously in the `dependentParameters` method.

For example: `m_vehicle = m_empty + m_payload`.

If the payload changes in the experiment, the vehicle mass is recalculated in the `dependentParameters` method.

Constants: Constants describe system-wide values that do not change during an experiment, e.g., the gravitation constant.

Dimensions, Scopes, and Data Types: For each element type available, there are various dimensions, scopes, and data types. The possible combinations are listed below.

elements \ combinations	dimension			scope		data type			
	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

Fig. 9-2 Dimensions, Scopes, and Data Types

9.4 Block Methods

The methods (type and number) available in CT basic blocks are pre-defined and cannot be modified by the user. Each method has a specific purpose, e.g., the calculation of derivations or outputs. The execution sequence of the methods is fixed; the methods are executed automatically. It is not necessary to use each method in a CT basic block.

The following methods are available in CT basic blocks:

init(): The `init()` method is called only at the beginning or restart of an experiment. The `init()` method can be used to specify code for initializing the block, e.g., to model the start-up behavior of a model or to initialize state variables (e.g., `resetContinuousState(x, 5.3)`). Initialization values derived from calculation statements have to be explicitly assigned using the `init()` method.

terminate(): The `terminate()` method is executed at the end of the experiment. The `terminate()` method can be used to specify code for finishing a block, e.g., to model the shutdown behavior of the system.

derivatives(): Ordinary differential equations (ODE) have to be specified in the `derivatives()` method. If the model structure changes during the simulation (e.g., in a model with moving masses that simulates static and dynamic friction), the structure change can be controlled with **the usual control structures** (`if(...)` `then...` `else...`).

update(): `update()` is executed in the granularity of the external communication interval `dT`. Values required only at this granularity (also communication with the experiment environment) can be calculated using this method.

directOutputs(): The `directOutputs()` method includes all output equations with direct pass-through that directly depend on inputs. As they directly depend on inputs that in turn may depend on `nondirectOutputs()`, this method is executed after `nondirectOutputs()`.

nondirectOutputs(): The `nondirectOutputs()` method includes all output equations with nondirect pass-through (i.e., those not directly depending on inputs).

dependentParameters(): Within the `dependentParameters` method, equations are specified for parameters that depend on other parameters. This method is only executed if a parameter has been changed during the simulation experiment (asynchronous execution when changed). This reduces the calculation time.

For example: `m_vehicle = m_empty + m_payload`.

The vehicle mass is recalculated in the `dependentParameters` method only if the payload changes in the experiment.

stateEvents(): Within the `stateEvents()` method, it is possible to model state- and time-dependent discontinuities. This method is evaluated at the end of each consistent integration step. Discrete state equations must be specified in the `stateEvents()` method.

events(): The `events()` method can be used to process asynchronous software and hardware interrupts. This method is not executed time-synchronously but asynchronously when the corresponding event occurs.

9.5 Computing Sequence

During the execution of a simulation, the methods contained in a CT block are triggered in different cycles. There are three general cycle intervals:

- the external communication interval `dT`
- the integration step size `h`

- The step size h/n depending on the internal integration method

External Communication Interval dT

The communication interval is not part of the model but is chosen only at run-time of the simulation. The following communication occurs during the dT cycle:

- communication between CT blocks and the experiment environment, e.g., stimulation and visualization
- communication between CT blocks and controller modules within a hybrid project
- communication between several CT (structure) blocks within a hybrid project if several integration methods are used
- calling the `update()` method

Integration Step Size h

The integration step size is not part of the model but chosen only at run-time of the simulation. During the h cycle, communication takes place between several continuous time blocks within a continuous time structural block. After the integration step has been executed across all blocks, the `stateEvents()` method is executed.

Each value transferred is numerically acknowledged and depends on the selected integration method. When simulating a highly dynamic model for which h has to be very small, the speed can be considerably increased by selecting a much higher value for dT than for h .

Step Size Depending on the Internal Integration Method: h/n

Other than the h cycle, the h/n cycle depends on the selected integration method; e.g., the Euler integration method uses the cycle time h/l while the Heun integration method uses $h/2$.

During the h/n cycle, the intermediate steps of the integration are calculated. As for the h cycle, communication takes place between the continuous time blocks of a continuous time structure block. The intermediate steps of the integration cannot be communicated to the outside.

Numerically, no discontinuities can be handled during this cycle since the `stateEvents()` method is not called during this cycle.

There is the following relationship between the different step sizes:

$$dT \geq h \geq h/n$$

The entire cycle of the various method calls is depicted in Fig. 9-3:

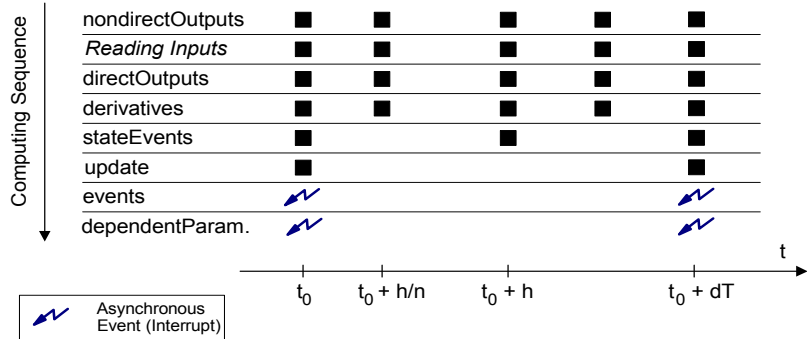


Fig. 9-3 Cycle of method calls in a continuous time block

The `events()` and `dependentParameters()` methods are only called when an explicit, asynchronous event occurs, and especially not during a dT cycle.

Fig. 9-4 shows the execution sequence of all methods from the start to the end of the simulation.

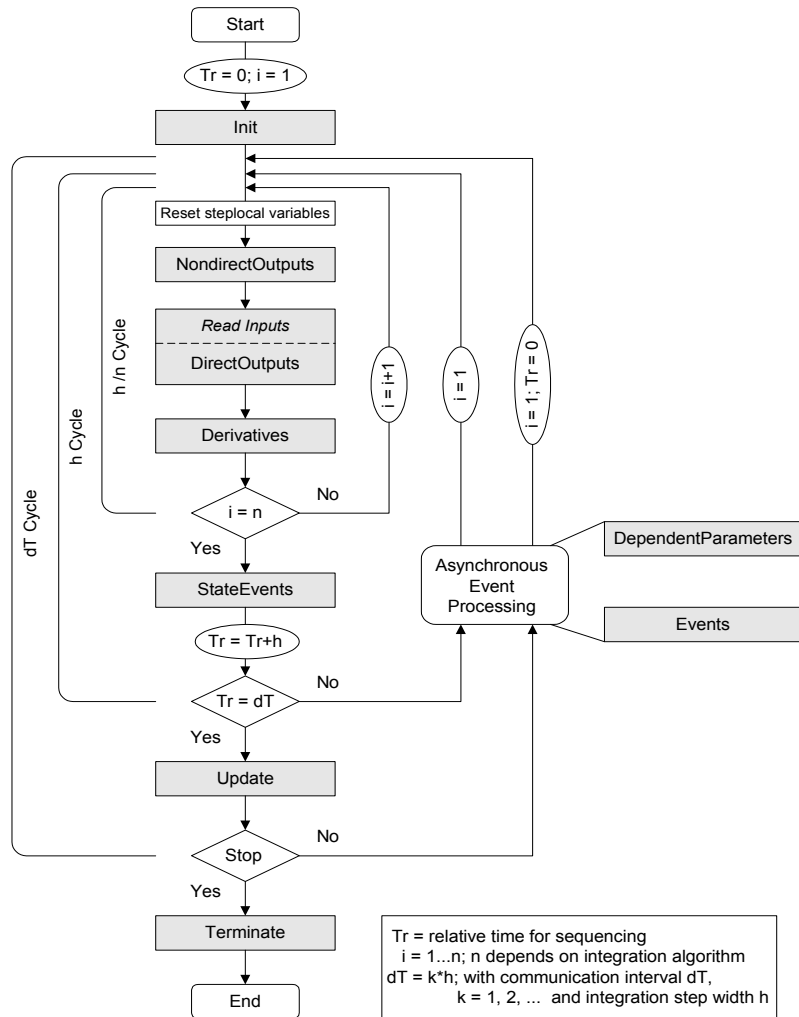


Fig. 9-4 Execution sequence of methods in the CT block

The sequence in which the methods of a basic block are executed is illustrated by means of the following examples.

The evaluation sequence for synchronous calls, e.g. if $n = 1$ (Euler) and $h = dT$, is:

- at time $t = dT$: nondirectOutputs - (reading inputs) - directOutputs - derivatives
- at time $t = dT$: stateEvents
- at time $t = dT$: update

For a more complex integration method, e.g., if $n = 2$ (Adams-Moulton) and $h = dT$, the sequence is:

- at time $t = dT/2$: nondirectOutputs - (reading inputs) - directOutputs - derivatives
- at time $t = dT$: nondirectOutputs - (reading inputs) - directOutputs - derivatives
- at time $t = dT$: stateEvents
- at time $t = dT$: update

The evaluation sequence for $n = 1$ and $h = dT/2$ is:

- at time $t = dT/2$: nondirectOutputs - (reading inputs) - directOutputs - derivatives
- at time $t = dT/2$: stateEvents
- at time $t = dT$: nondirectOutputs - (reading inputs) - directOutputs - derivatives
- at time $t = dT$: stateEvents
- at time $t = dT$: update

Understanding the computing sequence and thus the behavior of continuous time basic blocks is absolutely mandatory for a correct use of these blocks. Using ESDL as the modeling language gives the additional advantage of providing an automatic analysis phase that ensures consistent modeling when connecting several CT blocks. The computing sequence is especially important for blocks with direct outputs (directOutputs), because current values from the same iteration cycle have to be applied to the corresponding inputs.

9.6 Modeling with ESDL

The entire language scope of ESDL is available for the specification of continuous time basic blocks. In addition, a semantic check and a number of additional library functions for describing differential equations are provided. These are described in the following sections.

9.6.1 Differential Equations in ESDL

In ESDL, each continuous state variable supports the derivation operator `ddt`. Differential equations can be described with the `ddt` operator.

An example may be a PT2 system with the continuous state variables x and x_p , the input in , and the parameters d , T , K . The mathematical description of the system is:

$$\begin{aligned}x' &= x_p; \\x_p' &= (K \cdot in - (2.0 \cdot d \cdot T \cdot x_p) - x) / (T \cdot T); \end{aligned}$$

When modeling this PT2 system with ESDL, the derivations are specified by means of the `ddt` method:

```
x.ddt(xp);
xp.ddt( (K*in - (2.0*d*T*x.ddt()) - x) / (T*T) );
```

The derivatives on the left side of a differential equation (i.e., in the argument of a derivation method) cannot be accessed. If such an access is required, the system needs to be reformulated.

The `ddt` operator can only be used in the `derivatives()` method.

9.6.2 Semantic Checks in ESDL

Semantic checks can be performed when using ESDL within a continuous time method. The verification items ensure that the model matches the fundamental continuous time simulation framework. For example, it is not permitted to change the value of a state variable directly (instead, the `resetContinuousState()` function has to be used to internally reset the integration algorithm). Fig. 9-5 provides an overview of the access rights to those elements. The semantic check traps any violation of these rights.

method \ element	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 = read
w = write

Fig. 9-5 Access Rights to Elements

The derivation operator `dDt` supports only the first derivative. The output equations of the `nondirectOutputs()` method are analyzed to detect a direct dependency on an input. If such a case is found, a warning is issued.

9.6.3 Additional Library Functions

For advanced continuous time modeling with ESDL, the system library provides a number of additional library functions:

- `getTime()`
- `getdT()`
- `getIntegrationStepsize()`
- `resetContinuousState()`
- `resetCTSolver()`

The following describes the use of each library function in detail. Access to these functions in each method is shown in Fig. 9-6.

method	additional library functions				
	getTime	getdT	getIntegrationStepsize	resetContinuousState	resetCTSolver
init	+	+	+	+	+
derivatives	+	+	+	-	-
update	+	+	+	+	+
directOutputs	+	+	+	-	-
nondirectOutputs	+	+	+	-	-
terminate	+	+	+	-	-
events	+	+	+	-	-
dependentParameters	+	+	+	-	-
stateEvents	+	+	+	+	+

+ available
- not available

Fig. 9-6 Access to functions in the methods of a continuous time block

getTime(): In some cases, the current simulation time is of importance. For on-line experiments, this is the actually elapsed time. This value can be obtained using the `getTime` library function:

```
t = getTime ( );
```

The `getTime` function can be used in any method.

getdT(): The `getdT` library function provides the current step size for external communication:

```
step = getdT ( );
```

getIntegrationStepsize(): The `getIntegrationStepsize()` library function returns the current integration step size:

```
h = getIntegrationStepsize ( );
```

resetContinuousState(state, new value): Modeling time- or state-dependent discontinuities often requires resetting the continuous state variable. To ensure a correct numeric evaluation, the integration method needs to be reinitialized internally. This is done using the `resetContinuousState` function:

```
resetContinuousState (x, 0.0 );
```

In this case, the state x is set to 0.0 and, if necessary, the integration method is reinitialized. Use of the `resetContinuousState` library function is permitted only in the `init`, and `stateEvents` methods. Use of the function is also allowed in the method `update`, but it is useless because that method has no write access to continuous states. It is useless. `resetContinuousState(x,y)` is followed automatically by `resetCTSolver()`.

`resetCTSolver()`: With `resetCTSolver`, the integration method can be reset explicitly:

```
resetCTSolver ( );
```

Use of the `resetCTSolver` library function is permitted only in the `init`, `update`, and `stateEvents` methods. `resetContinuousState(x,y)` is followed automatically by `resetCTSolver()`.

9.7 Modeling in C

Modeling in C offers the capabilities of the C language but no semantic checks. Continuous time basic blocks specified in C may be hardware-dependent. If programming is done in ANSI-C, it is possible to create hardware-independent models even in C. This is necessary if pointers or C subroutines are to be used. C basic blocks can be used to model hardware-dependent blocks and in the same way as ESDL basic blocks. C basic blocks require an explicit specification whether they have a direct pass-through (output depends directly from the input) or an indirect pass-through by selecting `direct` or `nondirect` in the "*Block Behavior*" combo box. This affects the automatic determination of the execution sequence.

Note

When modeling in C, there are no semantic checks ensuring consistent modeling (as in ESDL). Consistency has to be ensured by the user. It is recommended to use C for modeling continuous time systems only if absolutely necessary, e.g., for modeling controller-dependent system portions or if C pointers or C subroutines have to be used.

9.7.1 Differential Equations in C

In C, an internal derivation variable is created for each continuous state variable. The name of this variable is composed of the name of the state variable and the prefix `dat`.

Examples are the continuous state variables x and x_p ; the automatically created derivation variables are `datx` and `datxp`. They are visible in all methods.

A complete example is a PT2 system with the continuous state variables x and x_p , the input i_n , and the parameters d , τ , κ .

```

x' = xp;
xp' = (K*in - (2.0*d*T*xp) - x) / (T*T);

```

The PT2 system above can be expressed as C code in the CT block as follows:

```

ddtx = xp;
ddtxp = (K*in - (2.0*d*T*ddtx) - x) / (T*T);

```

9.7.2 Additional C Routines

Additional C routines are available for modeling in C. For generic use of these routines, the internal data structure of the current block must be specified in the routine's interface. The `CTBlock` and `self` methods are visible in each method.

The following routines are provided:

- `getTime`
- `getdT`
- `getIntegrationStepsize`
- `resetCTSolver`
- `sizeU`
- `sizeY`
- `sizeV`
- `sizeX`
- `sizeXK`

The `get` and `reset` routines provide additional ESDL library routines; the `size` routine allows a generic model design if the number or array size of instance variables has to be changed.

The following describes the use of the additional C routines in more detail. There are no semantic checks and usage restrictions provided with these routines. It is the user's responsibility to ensure they are used correctly.

real64 getTime(CTSimExperiment *):

The `getTime` function returns the current simulation time:

```
t = getTime (CTBlock);
```

real64 getdT ():

The `getdT` function returns the current interval for external communication:

```
step = getdT ();
```

```
real64 getIntegrationStepsize(CTSimExperiment *):
```

The `getIntegrationStepsize` function returns the current integration step size:

```
h = getIntegrationStepsize (CTBlock);
```

```
void resetCTSolver(CTSimExperiment *):
```

The integration algorithm can be reset explicitly with the `resetCTSolver` routine. An example for its use is resetting a continuous time state:

```
x = 0.0;
resetCTSolver (CTBlock);
```

Whenever one or more continuous time states have been set explicitly, the internal structures need to be reset when finished. Note that the `resetCTSolver` command should always be issued after a value has been assigned to a continuous time state.

```
int_32 sizeU (CTSimExperiment *):
```

The `sizeU` function returns the number of block inputs:

```
sizeU = sizeU (CTBlock);
```

If some of the inputs are arrays, the total number of the scalar elements is returned. More complex inputs, such as records, structures or classes, are counted as one element.

```
int_32 sizeY (CTSimExperiment *):
```

The `sizeY` function returns the number of block outputs:

```
sizeY = sizeY (CTBlock);
```

If some of the outputs are arrays, the total number of the scalar elements is returned. More complex outputs, such as records, structures or classes, are counted as one element.

```
int_32 sizeV (CTSimExperiment *):
```

The `sizeV` function returns the number of block parameters (parameters and dependent parameters):

```
sizeV = sizeV (CTBlock);
```

If some of the parameter states are arrays, the total number of the scalar elements is returned.

```
int_32 sizeX (CTSimExperiment *):
```

The `sizeX` function returns the number of continuous states:

```
sizeX = sizeX (CTBlock);
```

If some of the continuous states are arrays, the total number of the scalar elements is returned.

int_32 sizeXK (CTSimExperiment *):

The `sizeXK` function returns the number of discrete states:

```
nofX = sizeXK (CTBlock);
```

If some of the discrete states are arrays, the total number of the scalar elements is returned.

10 **Continuous Time Structure Blocks and Graphical Hierarchies**

Continuous time structure blocks (CT structure blocks) can be used to build complex models by combining and linking other CT structure and CT basic blocks in a graphical block diagram. A slightly modified Block Diagram Editor (BDE) is provided for the specification of continuous time structure blocks (refer also to Fig. 8-2 on page 175). The corresponding inputs and outputs are graphically connected with each other in the BDE.

A continuous time structure block is modeled as a block diagram with a fixed number of methods. In principle, the methods of the CT basic blocks are automatically applied and cannot be modified in the BDE. The functional description is tied to a single diagram. The correct computing sequence is also determined automatically and cannot be influenced directly.

For a simple example illustrating the use of CT basic blocks, their methods, and CT structure blocks up to the simulation in the experiment environment, refer to the tutorial (volume "Getting Started", chapter "Modeling a Continuous Time System").

10.1 **Reuse of Structure Blocks**

CT structure blocks are stored in the database, the same as CT basic blocks, and are available for other CT structure blocks. This allows building a model library for a modular and hierarchical model structure. If a CT basic block or CT structure block is changed in the database, the change is automatically applied to all models in the database. Maintenance of the CT blocks is thus required at one place only.

Finished models whose CT blocks may not be replaced with newer versions, must therefore be stored in a different database.

10.2 **Elements of a Continuous Time Structure Block**

Not all variables that are used in basic blocks are required in continuous time structure blocks. The following elements are available:

- Inputs
- Outputs
- Global parameters
- Constants
- OneD and TwoD table parameters

For each element type, there are different dimensions, scopes, and data types.

Addition and subtraction operators are provided for which the number of inputs can be selected individually.

10.3 Block Interfaces

The following sections describe the elements available in continuous time structure blocks.

Inputs: Block entries are described by inputs. During each evaluation step, all input variables are read.

Outputs: Block exits are described by outputs. During each evaluation step, all output variables are updated.

Global Parameters: Global parameters are used to describe parameters that are visible in the entire model. A global parameter usually corresponds to a global characteristic property of the real system. An efficient use of global parameters can reduce the complexity and facilitate the maintenance of the model.

Constants: Constants are used for values that do not change during an experiment, such as the gravitation constant.

Dimensions, Scopes, and Data Types: Each type of element has a certain dimension, a scope of validity, and a type. The possible combinations are illustrated in the table below.

combinations elements	dimension			scope		data type			
	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
global parameter	x	x			x	x	x	x	x
constant	x	x		x	x	x	x	x	x

Fig. 10-1 Dimension, scope, and data type of elements

10.4 Operators

According to the systems theory, only linear operators are required for the description of structure blocks. Nonlinear elements are encapsulated in basic blocks. Therefore, only addition and subtraction operators are provided.

10.5 Algebraic Loops

In the following equation system

$$x = f_1(z)$$

$$y = f_2(x)$$

$$z = f_3(\text{input } a) \text{ (input } a \text{ assumed valid)}$$

each equation, with the exception of f_3 , depends on another equation. In order to allow the system to be computed from top to bottom correctly, the equations have to be rearranged as follows:

$$z = f_3(\text{input } a)$$

$$x = f_1(z)$$

$$y = f_2(x)$$

In this sequence, the system can be easily computed, even by conventional PC programs.

An algebraic loop exists if:

$$y = f_1(x) ;$$

$$x = f_2(y) ;$$

i.e., if two functions directly depend on each other. y is needed to calculate x , and x is needed to calculate y .

10.6 Direct and Nondirect Output

ASCET sorts CT blocks or methods in connected CT blocks directly depending on each other automatically in the correct order (automatic sequencing). If an algebraic loop exists in the model, ASCET terminates with an appropriate error message when determining the computing sequence. This occurs, for example, if two or more CT blocks with direct outputs form a feedback loop.

To enable the automatic determination and control of the computing sequence, the output property has to be specified. Outputs that directly depend on inputs have to be specified or described in the `directOutputs` method. Such a CT basic block is said to have a *direct output* or a *direct pass-through*. Outputs that do not directly depend on inputs are specified in the `nondirectOutputs` method. Such a CT basic block is said to have a *nondirect output* or a *nondirect pass-through*.

Wrongly declared outputs (e.g., direct output in the `nondirectOutputs` method) are detected if the ESDL modeling language is used. In CT blocks written in the C programming language, the *nondirect* or *direct* property is determined by the model designer.

The `nondirectOutputs` and `directOutputs` methods essentially determine the behavior of the CT basic blocks and the computing sequence in a CT structure block. This is illustrated again in the following example.

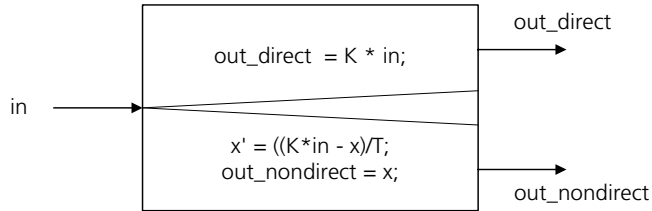


Fig. 10-2 Example: Direct and nondirect output

In general, an output has a direct pass-through behavior if it directly depends on one of the inputs. For example, an amplifier block (p behavior) is described by the function:

$$out = K * in$$

The output directly depends on the input. Consequently, the inputs have to be read first before the output can be calculated. The function must be written using the `directOutputs` method.

If the output does not depend on one of the inputs, for example, if the output depends on a continuous state or a parameter condition, it does not have direct pass-through behavior. Nondirect outputs are calculated from the values of the previous step. A CT block having a direct output terminates an existing loop. An example is the so-called PT_1 behavior:

$$x' = ((K * in - x) / T);$$

$$out = x;$$

The differential equation is solved using an integration method that requires the last output value and the input value `in` to calculate the current output value `x`. The assignment `out=x` has to be written using the `nondirectOutputs` method (the differential equation is discussed in the `derivatives` method).

Two simple examples illustrate a correct and an incorrect coupling of two CT basic blocks with direct and nondirect output within a CT structure block. It is essential to understand that a direct output requires the input data of the current time step. A nondirect output can be calculated and sent without the input information of the current time step. Therefore, the direct outputs are calculated after the nondirect outputs.

Fig. 10-3 shows a combination of two CT blocks with direct and nondirect pass-through behavior that does not cause an algebraic loop.

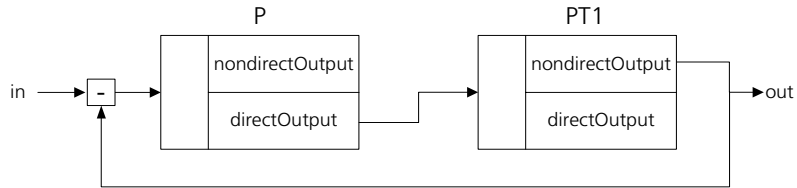


Fig. 10-3 Circuit of CT blocks with direct and nondirect outputs

The P block requires a valid input value for its calculation. Consequently, the `nondirectOutputs` method in the PT_1 block has to be calculated first and then the `directOutputs` method in the P block.

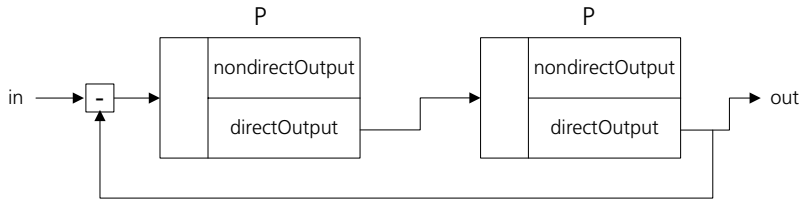


Fig. 10-4 Algebraic loop

In Fig. 10-4, two CT blocks with direct outputs are connected in series. This results in an algebraic loop. Each block requires the current output value of the other block. ASCET reports this error.

Direct pass-through circuits must be avoided. However, it is not possible to resolve algebraic loops automatically and implicitly because an implicit resolution of algebraic loops requires an iterative method, which is not acceptable under real-time conditions.

An advantage over the automatic resolution of an algebraic loop is that the user, knowing his model, can insert a block without a direct output at the most appropriate position so the subsequent blocks can be computed in the next iteration step.

In principle, there are two alternatives to avoid algebraic loops:

1. Inserting a block without a direct output. As this corresponds to a storage element, the integration step size may have to be decreased to avoid that the dynamics of the model is impaired.
2. Modifying the model structure to eliminate the algebraic loop. Reformulating the equations in the CT basic blocks, modifying the structure.

10.7 Difference Between Graphical Hierarchies and CT Structure Blocks

Externally, CT structure blocks behave like CT basic blocks regarding their computing sequence. The computing sequence is determined in the CT structure block. The structure behaves like a block with direct or nondirect output depending on whether the outputs of the structure block depend on the inputs directly or nondirectly.

Hierarchies, however, have a purely symbolic nature used to layout a CT structure block more clearly. They do not affect the simulation. Fig. 10-5 (left part) shows an example in which a CT structure block has a direct output to a CT basic block which in turn has a direct output to the same CT structure block. This causes an algebraic loop as the two blocks within the structure block are computed directly succeeding each other (virtually simultaneously). However, the second CT block within the structure block requires a current output of the external CT block.

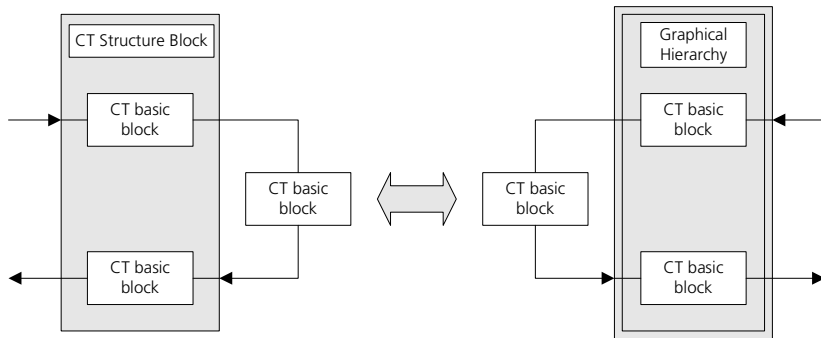


Fig. 10-5 Structuring with CT structure blocks or graphical hierarchies

The algebraic loop can be avoided by resolving the structure block and replacing it with a graphical hierarchy (Fig. 10-5, right-hand part) that combines model parts obviously related with each other. A drawback of hierarchies is that they cannot be stored separately but only together with the structure block in which they are contained.

10.8 Computing Sequence of Methods Within a Structure

The computing sequence in a CT structure block is essentially determined by the computing sequence of the methods within a CT basic block that was described in chapter 9.5 (refer to Fig. 9-4 on page 190). It depends mainly on the integration method and the selected time or communication intervals.

In principle, the methods in the structure block are computed in the same sequence as in the basic block (`init`, `nondirectOutputs`, `directOutputs`,...), with the same method being executed first in all basic blocks of the structure block before switching to the next method. This means that the `init` method is first executed in all blocks before starting the `nondirectOutputs` method in any basic block.

As long as the `directOutputs` method is not used in any CT basic block, the sequence is exclusively determined by the CT basic blocks. The order in which the same method is executed in the individual CT basic blocks is not important. This means that first all `init` methods are computed, then all `nondirectOutputs` methods, etc., each method in any arbitrary order of blocks.

If the `directOutputs` method is used in more than one block, the computing sequence becomes important, because some of the inputs of the `directOutputs` methods require current values from other outputs. If the input is connected to an output of the `nondirectOutputs` method, there is always a current value, because this method is first computed in all CT blocks before starting the `directOutputs` method. However, if the input depends on the output of another `directOutputs` method, this method must be computed first.

Example: Computing Sequence

Fig. 10-6 shows the computing sequence in a small CT structure block with coupled CT basic blocks. `readInputs` is not a method in its own right but belongs to `directOutputs`; it is shown to emphasize that current values have to be read first in order to compute the `directOutputs` method. The computing sequence is determined by the automatic sequencing algorithm. The numbers indicate the order of processing. Identical numbers mean that the execution sequence is arbitrary.

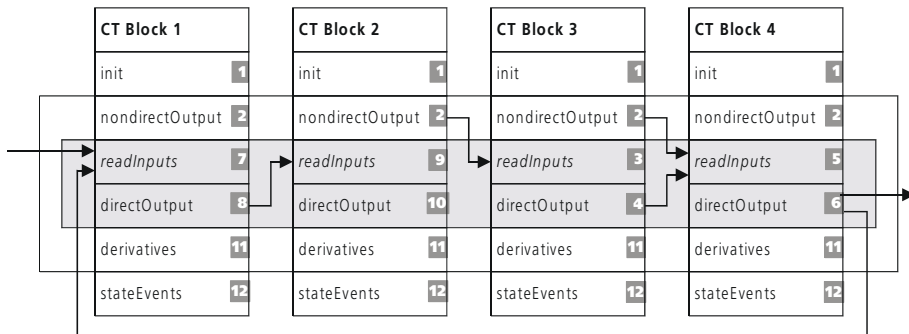


Fig. 10-6 Computing sequence of methods for coupled CT blocks

The computing sequence is especially important for CT blocks with direct outputs (`directOutputs` method), because current values from the same iteration cycle have to be applied to the corresponding inputs (shaded sections in Fig. 10-6).

The CT blocks are processed from top to bottom. Furthermore, each method is executed sequentially one after the other. `init` is executed only once at the start of the simulation.

Within the integration loop (`nondirectOutputs` up to `derivatives` methods), all `nondirectOutputs` are always computed. Their sequence is not fixed. As the `directOutputs` method directly depends on the corresponding input, ASCET searches all `directOutputs` methods until the corresponding `readInputs` no longer depends on another `directOutputs` method (shaded section in Fig. 10-6). In Fig. 10-6, this is the case in CT basic block 3. This results in the following sequence for reading the inputs and executing the `directOutputs` method:

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)

Only then the `derivatives` methods 1-4 are executed in arbitrary order. In case of a single-stage integration method, now follow the `stateEvents` methods for the CT blocks 1-4 in arbitrary order. Then again back to `nondirectOutputs`.

In case of n-stage integration methods, `nondirectOutputs` - `directOutputs` (as described above, in the correct order) and `derivatives` of CT blocks 1-4 are executed n times, before `stateEvents` is executed (also refer to Fig. 9-4 on page 190).

This means that the communication for combined CT basic blocks and/or CT structure blocks within one structure also occurs during the intermediate steps of the integration method. Each time, the `nondirectOutputs` up to `derivatives` methods are executed (single line frame).

The `update` method is executed after `stateEvents` only at the granularity of the communication interval `dT` and `terminate` only at the end of the simulation. For each, the computing sequence within the structure block is arbitrary.

There are therefore typically several equivalent computing sequences to solve a structure block. The sequencing algorithm of ASCET automatically selects one of the possible sequences.

Example: Execution Not Possible

If there is an algebraic loop, the computing sequence cannot be determined automatically. This situation is shown in Fig. 10-7. Each input of a `directOutputs` method depends on another `directOutputs`, closing the loop from CT block 4 to CT block 1. This results in an appropriate error message.

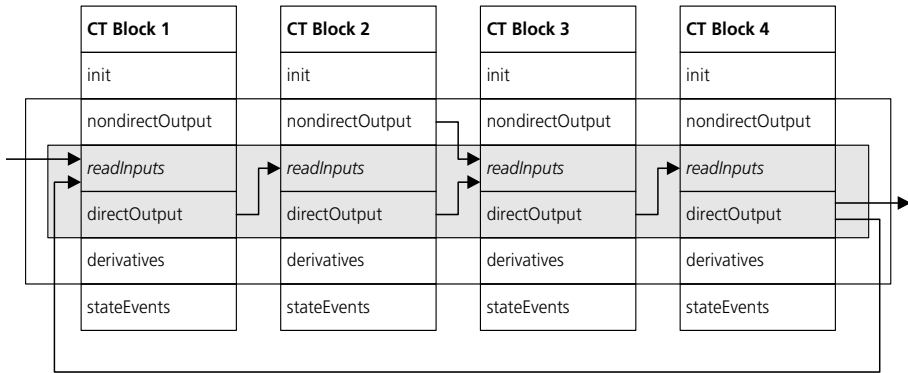


Fig. 10-7 Algebraic loop

11 Projects and Hybrid Projects

Projects are used for:

- Online simulation (hardware-in-the-loop) of CT blocks and Standard ASCET blocks
- Continuous time modeling of several CT structures with different integration algorithms and step sizes in a project
- Simulation of CT blocks in real-time

A project can consist of standard or/and continuous time modules or structures. A hybrid project is a project that contains both standard ASCET blocks and continuous time components. For example, in the hardware-in-the-loop simulation, the sending, receiving, and processing of signals from the real process (that is simulated by continuous time structures) are usually processed by standard modules.

When modeling and simulating systems with very fast and very slow components, e.g., hydraulic and mechanical components, the computing time can be reduced by using different integration methods or different integration steps. For this purpose, the respective model parts have to be located in a CT basic block or CT structure block, as appropriate.

The various CT model parts are loaded into a project and connected with each other in the Block Diagram Editor. In a project, each CT model part (CT basic block or CT structure block) can be computed as an independent process using a separate integration method and integration step size. It should be noted, however, that the individual blocks are linked to different tasks that communicate with each other only in fixed, selectable time intervals dT .

There is no exchange of values for intermediate steps of the integration method as is the case for coupling CT blocks with CT structures. There is also no automatic semantic verification as for CT structures that determines the computing sequence for the integration. The above applies only to CT blocks/structure at the project level. CT blocks and CT structures within the CT structures communicate at the granularity of the integration step size, of course, also within projects.

To ensure numeric stability, strongly cohesive systems should, therefore, not be coupled at the project level but within CT structures. Systems with weak cohesion can, however, be structured in projects. The advantage for weakly cohesive systems with highly disparate dynamic properties is that the integration method and integration step size can be selected individually to achieve an optimal computing time.

Fig. 11-1 schematically shows a project composed of one discrete standard block and two different CT structure blocks.

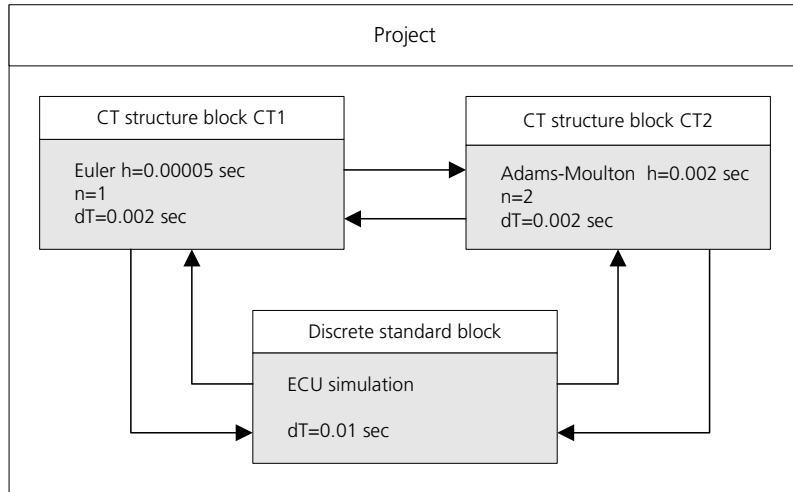
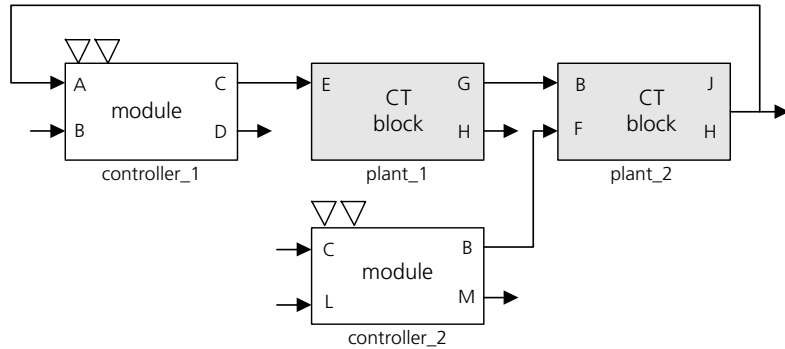


Fig. 11-1 Project with two continuous time blocks and one discrete block
 A hybrid project (e.g., for the ECU test automation) combines a controller model (Standard ASCET block) with a control system model (continuous CT structure blocks). The continuous time model part is itself composed of two CT structure blocks with different integration methods and different step sizes. The communication between the CT blocks takes place at 2 msec intervals while the CT blocks communicate with the discrete standard block every 10 msec.

11.1 Combining Continuous Time Blocks With Modules

Discrete modules in a project communicate via messages (global variables in ASCET blocks). There are no explicit connections (connecting lines) between Send and Receive messages; they are assigned to each other by their names.

Continuous time blocks, on the other hand, communicate among themselves and with modules via connections that have been specified graphically. The connections are built using the same method as in block diagrams.



implicit connection between modules:

controller_1 - B and controller_2 - B
 controller_1 - C and controller_2 - C

no implicit connection between modules and CT blocks

no implicit connection between CT blocks

Fig. 11-2 Combining continuous time blocks with modules

For discrete modules, the user has to explicitly define the tasks and to assign the processes defined in the module editor to the appropriate tasks.

CT blocks do not require an explicit definition of tasks, because these are defined automatically when needed. A `simulate` task and an `event` task are generated for each CT block. In addition, a common `init` task and a common `terminate` task are generated for all CT blocks in a project. For the example above, the following tasks are generated automatically:

- `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)`

These predefined tasks are static. They are all defined as cooperative tasks. The following sections describe the meaning of these tasks in more detail.

`simulate_CTn` **Tasks:**

For the `simulate_CTn` tasks, one simulation step is computed; the step size is ΔT . The step size can be specified for each `simulate_CTn` task individually; this allows for having several integration methods for different CT structure blocks within a project. The integration step size can be set during an experiment interactively. A simulation task normally uses the *Timer* trigger mode.

`event_CTn` **Tasks:**

When calling the `event_CTn` task, the `event` methods of the underlying CT blocks are executed. Because `event` methods are usually called asynchronously, the trigger mode of the `event_CTn` task should either be *Software* or *Event*.

`initialize_CT` **Task:**

When calling the `initialize_CT` task, the `init` methods of the underlying CT blocks are executed. As `init` methods are usually computed at the beginning of a simulation, the trigger mode of the `initialize_CTn` task should be *Init*.

`terminate_CT` **Task:**

When calling the `terminate_CT` task, the `terminate` methods of the underlying CT blocks are executed. The `terminate` task is automatically executed when the experiment finishes.

ASCET V5.2

Reference Lists

12 The ASCET System Library

12.1 Bit Operators

12.1.1 and



and returns the binary AND conjunction of the two arguments..

Methods	Arguments	Return Value
and	bitArray1:: unsigned discrete bitArray2:: unsigned discrete	unsigned discrete

On activation of method

and: The result of the binary AND conjunction of bitArray1 and bitArray2 is returned.

12.1.2 clearBit



clearBit resets the bit at the specified position of the argument. The position of the LSB¹ is 0.

Methods	Arguments	Return Value
clearBit	bitArray:: unsigned discrete position:: unsigned discrete	unsigned discrete

On activation of method

clearBit: The argument bitArray is returned with a zero-bit at position position.

¹. Least Significant Bit

12.1.3 `getBit`



`getBit` returns the value of the bit at the specified position of the argument as a logical value.

Methods	Arguments	Return Value
<code>getBit</code>	<code>bitArray::</code> unsigned discrete <code>position::</code> unsigned discrete	logical

On activation of method

`getBit`: `TRUE` is returned, if the bit at position `position` is equal to 1, otherwise `FALSE` is returned.

12.1.4 `or`



`or` returns the binary OR conjunction of the two arguments.

Methods	Arguments	Return Value
<code>or</code>	<code>bitArray1::</code> unsigned discrete <code>bitArray2::</code> unsigned discrete	unsigned discrete

On activation of method

`or`: The result of the binary OR conjunction of `bitArray1` and `bitArray2` is returned.

12.1.5 rotate



`rotate` rotates the bits of the argument to the left by a specified number of positions.

Methods	Arguments	Return Value
<code>rotate</code>	<code>bitArray::</code> unsigned discrete <code>k::</code> unsigned discrete	unsigned discrete

On activation of method

`rotate:` The result of the left-rotation of `bitArray` by `k` positions is returned.

12.1.6 setBit



`setBit` sets the bit at the specified position of the argument. The position of the LSB is 0.

Methods	Arguments	Return Value
<code>setBit</code>	<code>bitArray::</code> unsigned discrete <code>position::</code> unsigned discrete	unsigned discrete

On activation of method

`setBit:` The argument `bitArray` is returned with a one-bit at position `position`.

12.1.7 `shiftLeft`



`shiftLeft` shifts all bits of the argument to the left. The right bits are filled with zeros.

Methods	Arguments	Return Value
<code>shiftLeft</code>	<code>bitArray::</code> unsigned discrete <code>k::</code> unsigned discrete	unsigned discrete

On activation of method

`shiftLeft`: The result of the left-shift by `k` positions is returned. For `k=1` the result corresponds to the multiplication by two.

12.1.8 `shiftRight`



`shiftRight` shifts all bits of the argument to the right. The left bits are filled with zeros.

Methods	Arguments	Return Value
<code>shiftRight</code>	<code>bitArray::</code> unsigned discrete <code>k::</code> unsigned discrete	unsigned discrete

On activation of method

`shiftRight`: The result of the right-shift by `k` positions is returned.

12.1.9 toggleBit



toggleBit inverts the bit at the specified position of the argument.

Methods	Arguments	Return Value
toggleBit	bitArray:: unsigned discrete position:: unsigned discrete	unsigned discrete

On activation of method

toggleBit: The argument bitArray is returned with the bit at position k toggled.

12.1.10 writeBit



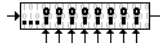
writeBit writes the value of the logical argument to the specified position of the unsigned discrete argument.

Methods	Arguments	Return Value
writeBit	bitArray:: unsigned discrete aBool::logical position:: unsigned discrete	unsigned discrete

On activation of method

writeBit For aBool = FALSE the argument is returned with a zero-bit at position position, for aBool = TRUE the argument is returned with a one-bit at position position.

12.1.11 writeByte



`writeByte` writes the values of eight logical inputs to the eight least significant bits of the argument.

Methods	Arguments	Return Value
<code>writeByte</code>	<code>bitArray::</code> unsigned discrete <code>b0::logical</code> <code>b1::logical</code> <code>b2::logical</code> <code>b3::logical</code> <code>b4::logical</code> <code>b5::logical</code> <code>b6::logical</code> <code>b7::logical</code>	unsigned discrete

On activation of method

`writeByte`: The argument is returned with the values of `b0` to `b7` written to the bit positions 0 to 7. 0 is the position of the LSB and the logical values `TRUE` and `FALSE` are mapped to 1 and 0 respectively.

12.1.12 xor



`xor` returns the binary exclusive OR conjunction of the two arguments.

Methods	Arguments	Return Value
<code>xor</code>	<code>bitArray1::</code> unsigned discrete <code>bitArray2::</code> unsigned discrete	unsigned discrete

On activation of method

`xor`: The result of the binary exclusive OR conjunction of `bitArray1` and `bitArray2` is returned.

12.2 Comparators

12.2.1 ClosedInterval



`ClosedInterval` returns `TRUE` if the value `x` is in the closed interval defined by `A` and `B`.

Methods	Arguments	Return Value
<code>out</code>	<code>x :: continuous</code> <code>A :: continuous</code> <code>B :: continuous</code>	logical

On activation of method

`out`: `TRUE` is returned, if $A \leq x \leq B$. Otherwise `FALSE` is returned.

12.2.2 LeftOpenInterval



`LeftOpenInterval` returns `TRUE` if the value `x` is in the left open interval defined by `A` and `B`.

Methods	Arguments	Return Value
<code>out</code>	<code>x :: continuous</code> <code>A :: continuous</code> <code>B :: continuous</code>	logical

On activation of method

`out`: `TRUE` is returned, if $A < x \leq B$. Otherwise `FALSE` is returned.

12.2.3 `OpenInterval`



`OpenInterval` returns `TRUE` if the value `x` is in the open interval defined by `A` and `B`.

Methods	Arguments	Return Value
<code>out</code>	<code>x :: continuous</code> <code>A :: continuous</code> <code>B :: continuous</code>	logical

On activation of method

`out` : `TRUE` is returned, if $A < x < B$. Otherwise `FALSE` is returned.

12.2.4 `RightOpenInterval`



`RightOpenInterval` returns `TRUE` if the value `x` is in the right open interval defined by `A` and `B`.

Methods	Arguments	Return Value
<code>out</code>	<code>x :: continuous</code> <code>A :: continuous</code> <code>B :: continuous</code>	logical

On activation of method

`out` : `TRUE` is returned, if $A \leq x < B$. Otherwise `FALSE` is returned.

12.2.5 GreaterZero



GreaterZero returns TRUE if the value x is greater than zero.

Methods	Arguments	Return Value
out	$x :: \text{continuous}$	logical

On activation of method

out: TRUE is returned, if $x > 0.0$. Otherwise FALSE is returned.

12.3 Counter & Timer

12.3.1 CountDown



CountDown decrements the counter and signals when the counter has reached zero.

Methods	Arguments	Return Value
start	startValue :: unsigned discrete	none
compute	none	none
out	none	logical

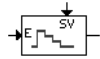
On activation of method

start: The counter is set to the start value.

compute: The counter is decremented by one.

out: TRUE is returned if the counter is greater than zero. Otherwise, FALSE is returned.

12.3.2 CountDownEnabled



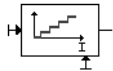
CountDownEnabled decrements the counter and signals when the counter has reached zero. This counter must be enabled explicitly.

Methods	Arguments	Return Value
start	startValue:: unsigned discrete	none
compute	enable::logical	none
out	none	logical

On activation of method

start:	The counter is set to the start value.
compute:	If enable is TRUE, the counter is decrement by one.
out:	TRUE is returned if the counter is greater zero. Otherwise, FALSE is returned.

12.3.3 Counter



Counter increments the counter by one.

Methods	Arguments	Return Value
reset	none	none
compute	none	none
out	none	unsigned discrete

On activation of method

reset:	The counter is set to zero.
compute:	The counter is increment by one.
out:	The counter value is returned.

12.3.4 CounterEnabled



Counter increments the counter by one. This counter must be enabled explicitly.

Methods	Arguments	Return Value
reset	initEnable:: logical	none
compute	enable::logical	none
out	none	unsigned discrete

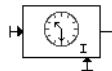
On activation of method

reset: If `initEnable` is TRUE, the counter is set to zero.

compute: If `enable` is TRUE, the counter is incremented by one.

out: The counter value is returned.

12.3.5 Stopwatch



StopWatch increments the time counter by one dT .

Methods	Arguments	Return Value
reset	none	none
compute	none	none
out	none	continuous

On activation of method

reset: The time counter is set to zero.

compute: The time counter is increment by dT .

out: The time counter value, i.e. the time elapsed since the last start, is returned.

12.3.6 StopwatchEnabled



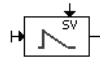
`StopwatchEnabled` increments the time counter by one dT . This timer must be enabled explicitly.

Methods	Arguments	Return Value
<code>reset</code>	<code>initEnable::logical</code>	none
<code>compute</code>	<code>enable::logical</code>	none
<code>out</code>	none	continuous

On activation of method

- `reset:` If `initEnable` is `TRUE`, the time internal counter is set to zero.
- `compute:` If `enable` is `TRUE`, the time counter is incremented by dT .
- `out:` The time counter value, i.e. the time elapsed since the last start and while `enabled` was `TRUE` is returned.

12.3.7 Timer



`Timer` decrements the time counter by dT and signals when the time counter has reached zero. It is not retriggerable.

Methods	Arguments	Return Value
<code>start</code>	<code>startTime::continuous</code>	none
<code>compute</code>	none	none
<code>out</code>	none	logical

On activation of method

- `start:` The time counter is set to `startTime` if the time counter value was previously less than or equal to zero.
- `compute:` The time counter is decremented by dT .

out : TRUE is returned, if the time counter value is greater than zero. Otherwise, FALSE is returned.

12.3.8 TimerEnabled



TimerEnabled decrements the time counter by dT and signals when the time counter has reached zero. It must be enabled explicitly.

Methods	Arguments	Return Value
compute	enable::logical in::logical startTime::continuous	none
out	none	logical

On activation of method

compute : If enable is TRUE, in has a rising edge and the time counter value is less or equal to zero, the timer is started, i.e. its counter value is set to the start time. Otherwise, the time counter is decremented by dT . If enable is FALSE, nothing happens.

out : TRUE is returned, if the time counter is greater than zero. Otherwise, FALSE is returned.

12.3.9 TimerRetrigger



TimerRetrigger decrements the time counter by dT and signals when the time counter has reached zero. It can be retriggered.

Methods	Arguments	Return Value
start	startTime::continuous	none
compute	none	none
out	none	logical

On activation of method

start: The time counter is set to the start value.
 compute: The time counter is decremented by dT .
 out: TRUE is returned, if the time counter value is greater than zero. Otherwise, FALSE is returned.

12.3.10 TimerRetriggerEnabled



TimerRetriggerEnabled decrements the time counter by dT and signals when the time counter has reached zero. It can be retriggered and must be enabled explicitly.

Methods	Arguments	Return Value
compute	enable::logical in::logical startValue::continuous	none
out	none	logical

On activation of method

compute: If enable is TRUE and in has a rising edge, the timer is started, i.e. its counter value is set to the start value. Otherwise, the time counter is decremented by dT (the time frame). If enable is FALSE, nothing happens.
 out: TRUE is returned, if the time counter value is greater than zero. Otherwise, FALSE is returned.

12.4 Delay

12.4.1 DelaySignal



DelaySignal delays its input signal by one evaluation step.

Methods	Arguments	Return Value
compute	signal::logical	none
out	none	logical

On activation of method

`compute`: The input signal is buffered.
`out`: The buffered signal is returned, thus the input signal is delayed by one step.

12.4.2 DelaySignalEnabled



`DelaySignalEnabled` delays its input signal by one evaluation step. It must be enabled explicitly.

Methods	Arguments	Return Value
<code>reset</code>	<code>initEnable::logical</code> <code>initValue::logical</code>	none
<code>compute</code>	<code>signal::logical</code> <code>enable::logical</code>	none
<code>out</code>	none	logical

On activation of method

`reset`: If `initEnable` is `TRUE`, `initValue` is buffered.
`compute`: If `enable` is `TRUE`, the input signal is buffered.
`out`: The buffered signal is returned, thus the input signal is delayed by one step.

12.4.3 DelayValue



`DelayValue` delays its input value by one evaluation step.

Methods	Arguments	Return Value
<code>compute</code>	<code>value::continuous</code>	none
<code>out</code>	none	continuous

On activation of method

compute:

The input value is buffered.

out:

The buffered value is returned, thus the input value is delayed by one step.

12.4.4 DelayValueEnabled



DelayValueEnabled delays its input value by one evaluation step. It must be enabled explicitly.

Methods	Arguments	Return Value
reset	initEnable:: logical initValue:: continuous	none
compute	value::continuous enable::logical	none
out	none	logical

On activation of method

reset:

If initEnable is TRUE, initValue is buffered.

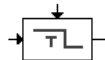
compute:

If enable is TRUE, the input value is buffered.

out:

The buffered value is returned, thus the input value is delayed by one step.

12.4.5 TurnOffDelay



TurnOffDelay delays a falling edge of the input signal.

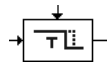
Methods	Arguments	Return Value
compute	signal::logical delayTime:: continuous	none
out	none	logical

On activation of method

compute : A falling edge of the input signal is delayed. If the signal flips from **TRUE** to **FALSE**, a timer is started. On being **FALSE** the timer is incremented by **dT** and is compared to **delayTime**. If the input signal is **TRUE**, the timer is reset.

out : **TRUE** is returned if the input signal is **TRUE** or the timer has not exceeded **delayTime**. Otherwise, **FALSE** is returned.

12.4.6 TurnOffDelayVariable



TurnOffDelay delays a falling edge of the input signal. The duration of the delay can be modified at runtime via the **Time** variable.

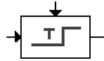
Methods	Arguments	Return Value
compute	signal ::logical delayTime ::continuous	none
out	none	logical

On activation of method

compute : A falling edge of the input signal is delayed. If the signal flips from **TRUE** to **FALSE**, a timer is started. On being **FALSE** the timer is incremented by **dT** and is compared to **delayTime**. If the input signal is **TRUE**, the timer is reset.

out : **TRUE** is returned if the input signal is **TRUE** or the timer has not exceeded **delayTime**. Otherwise, **FALSE** is returned.

12.4.7 TurnOnDelay



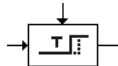
TurnOnDelay delays a rising edge of the input signal.

Methods	Arguments	Return Value
compute	signal::logical delayTime::continuous	none
out	none	logical

On activation of method

compute :	A rising edge of the input signal is delayed. If the signal flips from FALSE to TRUE , a timer is started. On being TRUE the timer is incremented by dT and is compared to delayTime. If the input signal is FALSE, the timer is reset.
out :	FALSE is returned if the input signal is FALSE, or the timer has not exceeded delayTime. Otherwise, TRUE is returned.

12.4.8 TurnOnDelayVariable



TurnOnDelayVariable delays a rising edge of the input signal. The duration of the delay can be modified at runtime via the Time variable.

Methods	Arguments	Return Value
compute	signal::logical delayTime::continuous	none
out	none	logical

On activation of method

compute :	A rising edge of the input signal is delayed. If the signal flips from FALSE to TRUE , a timer is started. On being TRUE the timer is incre-
-----------	--

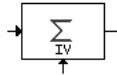
mented by `dT` and is compared to `delayTime`. If the input signal is `FALSE`, the timer is reset.

`out` :

`FALSE` is returned if the input signal is `FALSE`, or the timer has not exceeded `delayTime`. Otherwise, `TRUE` is returned.

12.5 Memory

12.5.1 Accumulator



`Accumulator` adds up its input value.

Methods	Arguments	Return Value
<code>reset</code>	<code>initValue::continuous</code>	<code>none</code>
<code>compute</code>	<code>value::continuous</code>	<code>none</code>
<code>out</code>	<code>none</code>	<code>continuous</code>

On activation of method

`reset` :

The accumulator value is set to `initValue`.

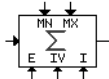
`compute` :

The accumulator is incremented by the input value, i.e. `accumulator (new) = accumulator (old) + input value`.

`out` :

The accumulator value is returned.

12.5.2 AccumulatorEnabled



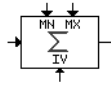
AccumulatorEnabled adds up its input value. It must be enabled explicitly and its accumulator value can be limited.

Methods	Arguments	Return Value
reset	initValue:: continuous initEnable:: logical	none
compute	value::continuous mn::continuous mx::continuous enable::logical	none
out	none	continuous

On activation of method

reset:	If <code>initEnable</code> is TRUE, the accumulator value is set to <code>initValue</code> .
compute:	If <code>enable</code> is TRUE, the accumulator is incremented by the input value, i.e. $\text{accumulator}(\text{new}) = \text{accumulator}(\text{old}) + \text{input value}.$ Additionally, the accumulator value is limited by <code>mn</code> and <code>mx</code> .
out:	The accumulator value is returned.

12.5.3 AccumulatorLimited



AccumulatorLimited adds up its input value. Its accumulator value can be limited.

Methods	Arguments	Return Value
reset	initValue:: continuous	none
compute	value:: <continuous </continuous mn:: <continuous </continuous mx:: <continuous< td=""> <td>none</td> </continuous<>	none
out	none	continuous

On activation of method

reset:	The accumulator value is set to <code>initValue</code> .
compute:	The accumulator is incremented by the input value, i.e. <code>accumulator(new) = accumulator(old) + input value</code> . Additionally, the accumulator value is limited by <code>mn</code> and <code>mx</code> .
out:	The accumulator value is returned.

12.5.4 RSFlipFlop



RSFlipFlop is a flip flop with a reset and a set input, where the reset input dominates the set input.

Methods	Arguments	Return Value
compute	r:: <logical </logical s:: <logical< td=""> <td>none</td> </logical<>	none
q	none	logical
nq	none	logical

On activation of method

compute :

If `r` is `TRUE`, the state of the flip flop is set to `FALSE`. Otherwise, if `s` is `TRUE`, the state is set to `TRUE`. If both `r` and `s` are `FALSE`, the state is left unchanged.

`q`:

The state of the flip flop is returned.

`ng`:

The negated value of the state is returned.

12.6 Miscellaneous

12.6.1 DeltaOneStep



`DeltaOneStep` returns the difference of the current input value and the last input value.

Methods	Arguments	Return Value
compute	<code>value::continuous</code>	none
out	none	continuous

On activation of method

compute :

The previous input value is subtracted from the input value.

out :

The difference is returned.

12.6.2 DifferenceQuotient



`DifferenceQuotient` computes the difference quotient of the input value.

Methods	Arguments	Return Value
compute	<code>value::continuous</code>	none
out	none	continuous

On activation of method

compute :

The difference quotient $(\text{value} - \text{previous value}) / \text{dT}$ is computed.

out :

The difference quotient is returned.

12.6.3 EdgeBi



EdgeBi detects a bidirectional edge of the logical input signal.

Methods	Arguments	Return Value
compute	signal::logical	none
out	none	logical

On activation of method

compute:	The input signal is compared to the previous input signal.
out:	TRUE is returned, if the input signal and the previous input signal differ. Otherwise, FALSE is returned.

12.6.4 EdgeFalling



EdgeFalling detects a falling edge of the logical input signal.

Methods	Arguments	Return Value
compute	signal::logical	none
out	none	logical

On activation of method

compute:	The input signal is compared to the previous input signal.
out:	TRUE is returned, if the input signal is low and the previous input signal was high. Otherwise, FALSE is returned.

12.6.5 EdgeRising



EdgeRising detects a rising edge of the logical input signal.

Methods	Arguments	Return Value
compute	signal::logical	none
out	none	logical

On activation of method

compute:	The input signal is compared to the previous input signal.
out:	TRUE is returned, if the input signal is high and the previous input signal was low. Otherwise, FALSE is returned.

12.6.6 Mux1of4



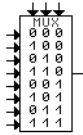
Mux1of4 switches between the four inputs values s_0, \dots, s_3 on the binary representation of their index.

Methods	Arguments	Return Value
out	b0::logical b1::logical s0::continuous s1::continuous s2::continuous s3::continuous	continuous

On activation of method

out:	The input value s_i (index i) is returned with $i = b_0 + 2*b_1$, interpreting FALSE as 0 and TRUE as 1.
------	--

12.6.7 Mux1of8



Mux1of8 switches between the eight inputs values s_0, \dots, s_7 on the binary representation of their index.

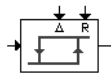
Methods	Arguments	Return Value
out	<code>b0::logical</code> <code>b1::logical</code> <code>b2::logical</code> <code>s0::continuous</code> <code>s1::continuous</code> <code>s2::continuous</code> <code>s3::continuous</code> <code>s4::continuous</code> <code>s5::continuous</code> <code>s6::continuous</code> <code>s7::continuous</code>	continuous

On activation of method

out : The input value s_i (index i) is returned with $i = b_0 + 2*b_1 + 4*b_2$, interpreting FALSE as 0 and TRUE as 1.

12.7 Nonlinears

12.7.1 Hysteresis-Delta-RSP



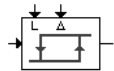
Hysteresis-Delta-RSP is a hysteresis with a right switching point and a delta offset

Methods	Arguments	Return Value
out	<code>x::continuous</code> <code>delta::continuous</code> <code>rsp::continuous</code>	logical

On activation of method

out: TRUE is returned, if $x > rsp$. FALSE is returned, if $x < (rsp - \delta)$. The return value is unchanged, if x lies within the open interval $] (rsp - \delta), rsp[$.

12.7.2 Hysteresis-LSP-Delta



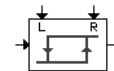
Hysteresis-LSP-Delta is a hysteresis with a left switching point and a delta offset.

Methods	Arguments	Return Value
out	$x::\text{continuous}$ $lsp::\text{continuous}$ $delta::\text{continuous}$	logical

On activation of method

out: TRUE is returned, if $x > (lsp + \delta)$. FALSE is returned, if $x < lsp$. The return value is unchanged, if x lies within the open interval $]lsp, (lsp + \delta)[$.

12.7.3 Hysteresis-LSP-RSP



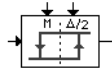
Hysteresis-LSP-RSP is a hysteresis with both a left and a right switching point.

Methods	Arguments	Return Value
out	$x::\text{continuous}$ $lsp::\text{continuous}$ $rsp::\text{continuous}$	logical

On activation of method

out: TRUE is returned, if $x > rsp$. FALSE is returned, if $x < lsp$. The return value is unchanged, if x lies within the open interval $]lsp, rsp[$.

12.7.4 Hysteresis-MSP-DeltaHalf



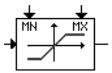
Hysteresis-MSP-DeltaHalf is a hysteresis with a middle switching point and a delta/2 offset.

Methods	Arguments	Return Value
out	x::continuous msp::continuous deltahalf::continuous	logical

On activation of method

out: TRUE is returned, if $x > (msp + \text{deltahalf})$. FALSE is returned, if $x < (msp - \text{deltahalf})$. The return value is unchanged, if input x is in the open interval $](msp - \text{deltahalf}), (msp + \text{deltahalf})[$.

12.7.5 Limiter



Limiter returns the input x limited by mn and mx .

Methods	Arguments	Return Value
out	x::continuous mn::continuous mx::continuous	continuous

On activation of method

out: The input x is limited by mn and mx and is returned, i.e. $\max(\min(x, mx), mn)$. There is no check if $mn \leq mx$.

12.7.6 Signum



Signum returns the sign of the input.

Methods	Arguments	Return Value
out	$x :: \text{continuous}$	continuous

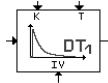
On activation of method

out: 1.0 is returned if $x > 0.0$, 0.0 is returned if $x = 0.0$, and -1.0 is returned if $x < 0.0$.

12.8 Transfer Function

12.8.1 Control

dT1



dT1 is a time discrete differentiation transfer function with time constant T and gain constant K .

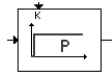
Methods	Arguments	Return Value
compute	$in :: \text{continuous}$ $T :: \text{continuous}$ $K :: \text{continuous}$	none
out	none	continuous

On activation of method

compute: The differentiation value is computed via a P-function and an I-function which is backcoupled.

out: The differentiation value is returned.

P



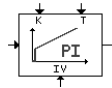
P is a time discrete proportional transfer function with gain constant K

Methods	Arguments	Return Value
out	in::continuous K::continuous	continuous

On activation of method

out : The return value $out = in * K$ is computed.

PI



PI is a time discrete proportional integrator with time constant T and gain constant K.

Methods	Arguments	Return Value
reset	initValue::continuous	none
compute	in::continuous T::continuous K::continuous	none
out	none	continuous

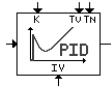
On activation of method

reset : The integrator value is set to `initValue`.

compute : The value of the PI-function is computed as the sum of a P-function and an I-function.

out : The value of the PI-function is returned.

PID



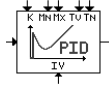
PID is a time discrete proportional integrator with differential part with time constants T_v and T_n and gain constant K .

Methods	Arguments	Return Value
reset	<code>initValue::</code> <code>continuous</code>	none
compute	<code>in::continuous</code> <code>Tv::continuous</code> <code>Tn::continuous</code> <code>K::continuous</code>	none
out	none	continuous

On activation of method

<code>reset:</code>	The integrator value is set to <code>initValue</code> .
<code>compute:</code>	The value of the PID-function is computed as a sum of a P-function, a D-function and an I-function.
<code>out:</code>	The value of the PID-function is returned.

PIDLimited



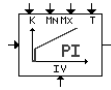
PIDLimited is a time discrete proportional integrator with differential part with time constants T_v and T_n and gain constant K . The value of the integrator is limited.

Methods	Arguments	Return Value
reset	initValue:: continuous	none
compute	in::continuous T_v ::continuous T_n ::continuous K ::continuous m_n ::continuous m_x ::continuous	none
out	none	continuous

On activation of method

reset:	The integrator value is set to <code>initValue</code> .
compute:	The value of the PID-function is computed as a sum of a P-function, a D-function and an I-function, where the integrator value of the I-function is limited by <code>m_n</code> and <code>m_x</code> .
out:	The value of the PID-function is returned.

PILimited



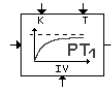
PILimited is a time discrete proportional integrator with time constant T and gain constant K . The value of the integrator is limited.

Methods	Arguments	Return Value
reset	initValue:: continuous	none
compute	in::continuous T::continuous K::continuous mn::continuous mx::continuous	none
out	none	continuous

On activation of method

reset:	The integrator value is set to <code>initValue</code> .
compute:	The value of the PI-function is computed as the sum of a P-function and an I-function, where the integrator value of the I-function is limited by <code>mn</code> and <code>mx</code> .
out:	The value of the PI-function is returned.

PT1



PT1 is a time discrete low pass with time constant T and gain constant K .

Methods	Arguments	Return Value
reset	initValue:: continuous	none
compute	in::continuous T::continuous K::continuous	none
out	none	continuous

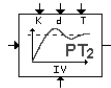
On activation of method

`reset` : The value of the integrator is set to `initValue`.

`compute` : The value of the PT1-function is computed via an I-function and a P-function which is back-coupled.

`out` : The value of the PT1-function is returned.

PT2



PT2 is a time discrete delay function with time constant T , gain constant K , and damping d .

Methods	Arguments	Return Value
<code>reset</code>	<code>initValue::continuous</code>	<code>none</code>
<code>compute</code>	<code>in::continuous</code> <code>T::continuous</code> <code>K::continuous</code> <code>d::continuous</code>	<code>none</code>
<code>out</code>	<code>none</code>	<code>continuous</code>

On activation of method

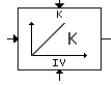
`reset` : The two integrator values are set to `initValue`.

`compute` : The value of the PT2-function is computed via two I-functions in row, which are backcoupled by a cascade of two P-functions.

`out` : the value of the PT2-function is returned.

12.8.2 Integrators

IntegratorK



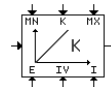
IntegratorK is a time discrete integrator with gain constant κ .

Methods	Arguments	Return Value
reset	initValue:: continuous	none
compute	in::continuous K::continuous	none
out	none	continuous

On activation of method

reset:	The integrator value is set to <code>initValue</code> .
compute:	The integrator value is computed via <code>integrator (new) = integrator (old) + in * dT* K</code> .
out:	The integrator value is returned.

IntegratorKEnabled



IntegratorKEnabled is a time discrete integrator with gain constant κ . It must be enabled explicitly and its integrator value can be limited.

Methods	Arguments	Return Value
reset	initValue:: continuous initEnable:: logical	none
compute	in::continuous K::continuous mn::continuous mx::continuous enable::logical	none
out	none	continuous

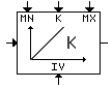
On activation of method

reset: If `initEnable` is `TRUE`, the integrator value is set to `initValue`.

compute: If `enable` is `TRUE`, the integrator value is computed via `integrator(new) = integrator(old) + in * dT * K` (limited by `mn` and `mx`).

out: The integrator value is returned.

IntegratorKLimited



`IntegratorKLimited` is a time discrete integrator with gain constant `K`. Its integrator value can be limited.

Methods	Arguments	Return Value
<code>reset</code>	<code>initValue::continuous</code>	<code>none</code>
<code>compute</code>	<code>in::continuous</code> <code>K::continuous</code> <code>mn::continuous</code> <code>mx::continuous</code>	<code>none</code>
<code>out</code>	<code>none</code>	<code>continuous</code>

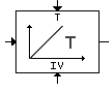
On activation of method

reset: The integrator value is set to `initValue`.

compute: The integrator value is computed via `integrator(new) = integrator(old) + in * dT * K` (limited by `mn` and `mx`).

out: The integrator value is returned.

IntegratorT



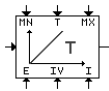
IntegratorT is a time discrete integrator with time constant T.

Methods	Arguments	Return Value
reset	initValue:: continuous	none
compute	in::continuous T::continuous	none
out	none	continuous

On activation of method

reset:	The integrator value is set to initValue.
compute:	The integrator value is computed via $\text{integrator}(\text{new}) = \text{integrator}(\text{old}) + \text{in} * \text{dT} / \text{T}$.
out:	The integrator value is returned.

IntegratorTEnabled



IntegratorTEnabled is a time discrete integrator with time constant T. It must be enabled explicitly and its integrator value can be limited.

Methods	Arguments	Return Value
reset	initValue:: continuous initEnable:: logical	none
compute	in::continuous T::continuous mn::continuous mx::continuous enable::logical	none
out	none	continuous

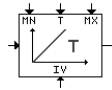
On activation of method

`reset`: If `initEnable` is `TRUE`, the integrator value is set to `initValue`.

`compute`: If `enable` is `TRUE`, the integrator value is computed via `integrator(new) = integrator(old) + in * dT / T` (limited by `mn` and `mx`).

`out`: The integrator value is returned.

IntegratorTLimited



`IntegratorTLimited` is a time discrete integrator with time constant `T`. Its integrator value can be limited.

Methods	Arguments	Return Value
<code>reset</code>	<code>initValue::continuous</code>	<code>none</code>
<code>compute</code>	<code>in::continuous</code> <code>T::continuous</code> <code>mn::continuous</code> <code>mx::continuous</code>	<code>none</code>
<code>out</code>	<code>none</code>	<code>continuous</code>

On activation of method

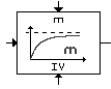
`reset`: The integrator value is set to `initValue`.

`compute`: The integrator value is computed via `integrator(new) = integrator(old) + in * dT / T` (limited by `mn` and `mx`).

`out`: The integrator value is returned.

12.8.3 Lowpass

DigitalLowpass



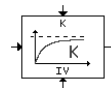
DigitalLowpass recursively computes the mean value of the input value.

Methods	Arguments	Return Value
reset	initValue:: continuous	none
compute	in::continuous m::continuous	none
out	none	continuous

On activation of method

reset: The mean value is set to `initValue`.
compute: The mean value is computed via mean value (new) = mean value (old) + $m \cdot (in - \text{mean value (old)})$.
out: The mean value is returned.

LowpassK



LowpassK is a simplified PT1-function with gain constant κ (low pass filter).

Methods	Arguments	Return Value
reset	initValue:: continuous	none
compute	in::continuous K::continuous	none
out	none	continuous

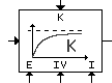
On activation of method

`reset`: The lowpass value is set to `initValue`.

`compute`: The lowpass is computed via `lowpass (new) = lowpass (old) + (in - lowpass (old)) * dT * K`.

`out`: The lowpass value is returned.

LowpassKEnabled



`LowpassKEnabled` is a simplified PT1-function with gain constant K (low pass filter). It must be enabled explicitly.

Methods	Arguments	Return Value
<code>reset</code>	<code>initValue::continuous</code> <code>initEnable::logical</code>	none
<code>compute</code>	<code>in::continuous</code> <code>K::continuous</code> <code>enable::logical</code>	none
<code>out</code>	none	continuous

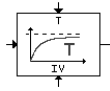
On activation of method

`reset`: If `initEnable` is TRUE, the lowpass value is set to `initValue`.

`compute`: If `enable` is TRUE, the lowpass is computed via `lowpass (new) = lowpass (old) + (in - lowpass (old)) * dT * K`.

`out`: The lowpass value is returned.

LowpassT



LowpassT is a simplified PT1-function with time constant T (low pass filter).

Methods	Arguments	Return Value
reset	initValue:: continuous	none
compute	in::continuous T::continuous	none
out	none	continuous

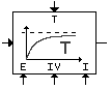
On activation of method

reset: The lowpass value is set to `initValue`.

compute: The lowpass is computed via `lowpass (new) = lowpass (old) + (in - lowpass (old)) * dT / T`.

out: The lowpass value is returned.

LowpassTEnabled



LowpassTEnabled is a simplified PT1-function with time constant T (low pass filter). It must be enabled explicitly.

Methods	Arguments	Return Value
reset	initValue:: continuous initEnable:: logical	none
compute	in::continuous T::continuous enable::logical	none
out	none	continuous

On activation of method

`reset :` If `initEnable` is `TRUE`, the lowpass value is set to `initValue`.

`compute :` If `enable` is `TRUE`, the lowpass is computed via $\text{lowpass}(\text{new}) = \text{lowpass}(\text{old}) + (\text{in} - \text{lowpass}(\text{old})) * \text{dT} / \text{T}$.

`out :` The lowpass value is returned.

13 Troubleshooting

In this chapter potential problems when working with ASCET are discussed and hints for solving these problems are given. If you have any problems that are not included in this chapter, please inform ETAS, so that this section can be enhanced.

In general, any system error indicated by ASCET may be a serious error, i.e. it is advisable to store all data to the database after a system error. If the system behavior is unexpected after a system error, the system error has caused an inconsistency in the running system. In this case you should leave ASCET and reboot Windows.

13.1 General Hints

Limit of the size of a database: The size of an ASCET database is limited to 4 GByte, the size of a single object to 128 MBytes. Be careful not to reach this limit when working with a large database, because when the limit is exceeded the database will be destroyed. Use the database tools to compact your database when necessary.

Conversion of databases: Databases that have been developed with the predecessor versions ASCET-SD V4.1 or V4.2 or ASCET V5.0 are automatically converted to ASCET 5.2. Note that the converted database cannot be used with older versions of ASCET. A backup copy of the old database is created automatically during conversion.

Databases created with ASCET-SD 4.0 or even earlier versions *cannot* be opened with ASCET 5.2.

ASCET supports only ANSI C compliant names. To ensure compatibility, you have to adjust the names of all items in the database using the built-in conversion tool. Choose **Tools → Database → Convert → All Names To ANSI C** in the Component Manager to convert the names of all items.

Problems with Graphics Cards: When problems with the displaying of ASCET windows appear, there is probably an incompatibility between ASCET, the graphics card and the graphics cards driver. When such problems occur, either try the most recent driver for your graphics card (which is usually available on the Internet from the card manufacturer) or try another resolution of your card. All standard VGA and SVGA modi should generally work.

The offline experiment runs out of time. The time (ΔT) for offline experiments has a limitation of approx. 3 days (in units of ΔT), i.e. if the ΔT is set very high (for instance 1000 seconds), the offline experiment will crash after a few minutes.

Unpredictable effects when using complex assignments: Unpredictable effects with the measuring of complex elements occur when complex assignments are executed in the model. A complex assignment is represented by an assignment of the respective pointers of the complex elements, that is, both objects are identical afterwards and one object is 'lost'. E.g. in the assignment $A=B$, the element A becomes the element B. The measurement and calibration system however still refers to both as separate objects. You can measure and calibrate the 'lost' object (here object A) but this has no effect and does not take into account the object that represents the complex element after the assignments (i.e. object B).

The fonts are not displayed properly: The Arial font family is not displayed properly under Win97/WinNT, so that some entries are hardly visible. Use the Microsoft SansSerif Font of size 10 instead. With this font, there are no display problems.

Problems with the external experimental targets: A potential source of errors when using the Centronics link cable is that the speed of the parallel port may be too fast for the Centronics link cable (esp. when using a Pentium 200 or higher). Here it is advisable to reconfigure the parallel port in the setup of the computer BIOS.

Busy ASCET: While ASCET is busy (e.g. generating code, committing to the database), do not try to invoke other functions in ASCET, but wait until the current action of ASCET is finished. Otherwise, the system behavior of ASCET may lead to unexpected errors (e.g. system errors).

13.2 Problems with ASCET

Some ASCET experiments do not end or do not run properly: Here the problem often lies with the C code that has been integrated into an ASCET model. Potential errors are wrong passing of parameters (when converting the ASCET type `continuous` the C type `double float` should be chosen), and infinite loops in the C code. Infinite loops may also occur in recursive object structures. A possible way to find the error here, is to exclude the C code components.

The generated code may not run in the scheduled time frame, i.e. its execution time is too long. Here either the specification must be changed, or a time frame with a longer interval should be assigned.

Another source of errors in this field is that sequence calls are not set properly or are simply forgotten.

The compilation returns unexplainable error messages or does not end: If you click into another window during compilation the priority for the DOS-box where the compilation takes place is decreased dramatically, so that the compilation comes to an almost complete stop. In that case you can activate the DOS-box by double clicking on its icon.

Additionally you should avoid the following keywords, which are used by the error management system to trace back compiler errors to the ASCET model: Error, ERROR, Serious, Fatal, illegal, Failed, failed, warning, known format.

ASCET does not compute correctly when using temporary variables: Automatic temporary variables can be used when the result of an expression is to be used in several different branches. These temporary variables are only computed once (upon evaluation of the first branch). When the branches using the temporary variable are only computed conditionally (e.g. as they are input to a switch or a MUX operator), the value of that temporary variable may not be computed correctly. Therefore automatic temporary variables should not be used, if the branches leading from a temporary variable are fed into a conditional operator.

L1-Communication Errors often occur during online experiments: In this case the priority of the communication process is too low. The priority of this process can be raised for the target in the file `es1130cp.inv`, `es1130cp_gnu.inv` or `es1135cp_gnu.inv` in the respective target directory. The file you have to edit depends on your target/compiler combination.

This file is used in the configuration of the compiler. Here you can modify the priority of the communication process by setting the parameter `__L1_Prio =` to the desired priority (by default it has the lowest priority, i.e. 0).

The documentation generation in .rtf format does not work properly. When displaying .rtf files, Word for Windows may not display the integrated bitmap image files. You may have to update all links to (external) * .gif files to view the images.

14 Code Generation Messages

This chapter contains the warnings and error messages that may appear during ASCET code generation, together with hints and explanations on how to correct the mistakes that led to the error. Error messages point to serious faults in the specification that lead to the code generation process to be terminated. Warnings point to less serious faults. The code generation process may be successful, but the resulting code may not work as desired.

14.1 Components

14.1.1 Error Messages

method <method_name> must be defined; need a return value

Description:

A method with return value has been declared in the component, but the return value does not have a sequence call attached to it. This is required, because the method might be called by other components.

Solution:

Edit the sequence call and select the method the return value belongs to as the sequence name. The sequence number must be the highest number attached to that method.

<method_name> has no argument <argument_name>

Description:

An operation attached to the method `method_name` uses an argument belonging to another method. A method may only use the local and global elements and its arguments, but not the arguments of other methods.

Solution:

Change the sequence call or replace the argument with another element.

missing argument connection for method <method_name> at block <block_name>

Description:

At the block `block_name` the method `method_name` is called, but not all arguments are connected, i.e. one of the arguments is missing. In the case of an operator, the method name is left blank.

Solution:

Connect the missing arguments, or in the case of an operator, choose an operator with the appropriate number of arguments.

double sequence number <sequence_number> **for** <name>

Description:

The process, method, action, or condition name has two sequence calls attached to it with the same sequence number `sequence_number`.

Solution:

Change one of the sequence numbers to a sequence number not yet used in name.

return value does not belong to <name>

Description:

A return value of some method or condition is assigned a sequence call belonging to a method or action name, which has no return value. The sequence call of a return value must always be assigned to the method or condition defining that return value.

Solution:

Change the sequence name of the sequence call of the return value to the name of the condition or method the return value belongs to.

delay-free loop detected at <block_name> **block**

Description:

A loop is created without any operation in that loop, e.g. the return value of an operator is directly fed in as an input to that operator.

Solution:

Insert an element into the loop.

type mismatch: expected <type_A>, **got** <type_B>

Description:

An argument of `type_B` is used where an argument of `type_A` is required, and the `type_B` can not be cast to `type_A`. E.g. an argument of type `cont` is fed into a logical operator. Presumably the connection is wrong.

Solution:

Supply an argument of the correct type.

type mismatch: expected <type_A> [`<name_A>`], **got** <type_B> [`<name_B>`]

Description:

An element with name `name_B` of `type_B` is assigned to a variable with name `name_A` of type `type_A` where `type_B` can not be cast to `type_A`. E.g. an element of type `cont` is assigned a variable of type `logical`. Presumably the connection is wrong.

Solution:

Change the type of the element or make a correct connection.

return must be the last operation of <name>

Description:

A method with a return value or condition `name` has a return statement whose sequence call does not have the highest sequence number in sequence calls attached to the method or condition.

Solution:

Change the sequence number in the sequence call to the highest number in all sequence calls belonging to the method or condition `name`.

<then> **part of IF block must be specified**

Description:

An IF block is used where THEN part is not used.

Solution:

Specify the THEN part. There must be at least one sequence call with a connector attached to the THEN part.

state machine needs start state

Description:

The state machine has no start state.

Solution:

Specify one of the states of the state machine as its start state.

multiple prio <priority_number> **for trigger** <trigger_name>
in state <state_name>

Description:

The state machine contains two transitions leading from state `state_name` attached to the same trigger `trigger_name` with the same priority `priority_number`. This is not allowed, since the transition is not unique.

Solution:

Change one of the priorities, such that all priorities leading from the same state and assigned to the same trigger are different.

unbalanced number of start/stop atomic in <name>

Description:

The method, process, condition or action `name` has sequence calls with attached atomic marks. However, there is an unbalanced number of start and stop marks.

Solution:

Insert or delete some of the start or stop marks, such that their number and appearance is balanced.

Expected consistent datamodel for <element_name> in <Class_name>. Element needs GET/SET direct access - please change attributes OR restore diagram

or—for ESDL/C code

method "<Element_name>"/"<function_name>" not defined as public in class "<Class_name>"

Description:

The element / function in the class <Class_name> has not been set for direct access/ made public.

Solution:

Enable direct access (Set/Get functionality) for the element or make the function public.

14.1.2 Warnings

<name> **not defined**

Description:

The method, process, or action has been declared, but was not defined. There is no sequence call with sequence name name. This only relates to methods without return values.

Solution:

Define the method, process or action or delete its declaration from the component interface.

type mismatch with casting from <type_B> [<name_B>], got <type_A> [<name_A>]

Description:

An element with name name_B of type type_B is assigned to a variable with name name_A of type type_A where a type cast is made from type_B to type_A. E.g. an element of type cont is assigned a variable of type sdisc.

Solution:

Change the type of the element or make a correct connection.

argument <argument_name> **of method** <method_name> **not used**

Description:

In the definition of the method `method_name` the argument `argument_name` of the method is not used.

Solution:

Use the argument `argument_name` in the definition of the method or delete it from the method definition.

unreachable state <state_name>

Description:

The state machine contains a state with name `state_name` that can not be reached from the start state, i.e. no transition leads to that state.

Solution:

Delete the state or make the state reachable from the start state.

literal value <value> **does not fit type** <type> **- limited to** <range_value>

Description:

The value of the literal is too large for the variable of type `type`, it is assigned to. The value of the literal for this assignment is automatically limited to the value `range_value`. This does not apply to expressions consisting of literals only. The type `type` is either `udisc` or `sdisc` which have a range of a 32 bit integer (unsigned or signed).

14.2 Projects

14.2.1 Error Messages

need binding for imported element <element_name>

Description:

The imported element or message `element_name` is not bound to a global element or message.

Solution:

Adjust the binding (either automatically or manually).

application modes missing for task <task_name>

Description:

The task `task_name` has no application mode assigned to it.

Solution:

Assign an application mode, or delete the task `task_name`. To exclude certain tasks from execution, simply specify an additional application mode with name `unused` and assign it to the tasks that are to be excluded.

14.2.2 Warnings

no start application mode specified - using `<opmode_name>`

Description:

None of the application modes is defined as the start mode. The application mode `opmode_name` is automatically defined as the start mode.

Solution:

Define one of the modes as the start mode, unless the right mode has been picked as the default.

missing trigger event

Description:

One of the event tasks specified in the operating system has no trigger event assigned to it.

Solution:

Change the mode of that task or assign one of the trigger events to that task.

14.3 Fixed Point Code Generation

14.3.1 Error Messages

Integer interval [a,b] of variable <name> too large for implementation type

Description:

The integer interval `[a,b]` derived from the model interval is too large for the chosen implementation type. Presumably, the implementation for this element has not been edited or the implementation type is not set to an integer type.

Solution:

Edit the implementation for the element `name`.

Cannot generate fixed point code for the non-linear formula <formula_name> of variable <name>

Description:

The non-linear formula `formula_name` is assigned to `name`. The fixed point code generation only supports linear formulae.

Solution:

Change the formula assigned to `name` or change the formula `formula_name`, so that it is a linear formula.

Physical interval [a,b] of divisor contains zero

Description:

Fixed point code can not be generated, because a division by zero could occur. This would result in an implementation interval of infinite size.

Solution:

Insert a variable for the divisor and specify a meaningful implementation for it (the physical interval should not contain zero).

14.3.2

Warnings

formula in implementation for <name> not known in current project - using default

Description:

In the implementation for the element `name` the formula is not known in the context of the current project. Presumably, no formula has been assigned. The identity formula is used instead.

Solution:

Use a valid formula from the context of the current project for the implementation for the element `name`.

Interval mismatch in assignment of <variable_name>: [a,b] := [c,d] (will be limited)

Description:

The fixed point code generator has found, that in the assignment of variable `variable_name` there is a possible conflict. The value of the expression that is assigned to the variable lies within the interval `[c,d]`. This interval is computed via interval arithmetics from the intervals specified for the elements in that expression. The interval `[a,b]` for the variable `variable_name` does not, however, include the interval `[c,d]`, so that an overflow might occur. To avoid this overflow, the value of the expression is automatically limited to the value interval of variable `variable_name` before the assignment is carried out. Note, that this warning cannot be avoided when there are arithmetic loops.

Index

Symbols

! 117
- 117
-- 117
!= 117
% 117
%= 118
&& 117
* 117
*= 118
+ 117
++ 117
+= 118
/ 117
/* comments */ 117
// comments 117
/= 118
< 117
<= 117
-= 118
== 117
> 117
>= 117
? : 118

|| 117

A

Abs operator 154
abs() 116, 137
access macros 170

- access to private methods 171
- array length 171
- arrays in external C code 171
- ASD_GET 171
- ASD_LENGTH 171
- ASD_RELEASE 171
- ASD_RESERVE 171
- ASD_SET 171
- ASD_USE_ARRAY_EXTERNAL 171
- direct access 171
- resource access 171
- self 171

accessing objects 124

- access control 126
- block diagrams 138
- C code access macros 170

accessing objects
 direct access methods 127
 library functions 136
 this 126

acos() 137

Action 45

 see also *state machine*

Adams-Moulton 181

AND 151

 see logical operators

application mode 19

argument 113, 166

arithmetic operator 117, 151

arrays 91, 127, 146

 access in ESDL 128

 Get/Set Port 147

 maximum size of ~ 128

 public interface 128

 Table Editor 128

asin() 137

assignment 114

 shorthand assignment operator 118

atan() 137

atomic sequences 141

auto-inlining 73

B

basic language elements 112

basic types 89

Between operator 154

between() 116

block diagram

 ~ vs. ESDL 141

 access ~ in ESDL 138

 semantics 159

block statements 114

branching

 see *control flow*

break 155

break 122, 124

C

C code

 access macros 170

 argument 166

 characteristic lines/maps 167

 direct access methods 171

 external ~ 169

 function parameters 163

 header 169

 local variables 166

 message 165

 method 162

 process 162

 specification 161

 variables 163

C programming language

 see programming languages

case operator 153

ceil() 137

characteristic line 92

 see also *one-dimensional table*

characteristic map 92

 see also *two-dimensional table*

characteristic table 92, 148

class 24

 hierarchical structure 31

 interface 27

 state machine 83

 vs. module 24

comment 116

 ~ in generated code 117

communication

 between processes 20

 message 20

comparison operator 151

complex element 100

component 23

 definition 25

 instantiation 25

 interface 27

 reusing 29

 specification 23

composite data types 127–134

 array 127

 data structures 134

 distributions 133

 group tables 133

 matrix 129

 one-dimensional table 129

 two-dimensional tables 131

compound statements 114

Condition 45

conditional construction

 see *control flow*

- conditional operator 118, 152
- constant 96, 116
 - system ~ 97
- cont 115
- Continuous Time block
 - see *CT block*
- Continuous time models
 - structure 173
- control flow 32, 120–124
 - break 122, 124
 - for 123
 - if...else 120
 - return 125
 - switch...case...default 121
 - while 123
- conventions
 - method names 112
 - variable names 114
- conversion
 - of data types 115
- cooperative scheduling 15
- cos() 137
- cosh() 137
- coth() 137
- csh() 137
- CT basic block 174, 183–198
 - interfaces 185
 - methods 186
- CT block 173–181
 - computing sequence 187
 - direct output 201
 - input 174
 - modeling in C 195
 - nondirect output 201
 - output 174
 - parameter 174
 - predefined tasks 211
 - state 174
 - structure 199
- CT structure block 174, 199–207

D

- data 101
 - transformation 109
- data set 101
- data structures
 - modelling ~ in ESDL 134

- data types
 - array 127, 146
 - basic ~ 115–116
 - composite ~ 127–134
 - continuous 115
 - conversion 115
 - data structures 134
 - distributions 133
 - group tables 133
 - logical 115
 - matrix 129, 146
 - messages 135
 - one-dimensional table 129
 - signed discrete 115
 - strings 112
 - two-dimensional tables 131
 - unsigned discrete 115
- diagram
 - item 143
 - line 143
 - pin 143
- diagram item 143
- differential equation 177
 - in C 195
- direct access methods
 - C code 171
 - ESDL 127
- distributions 133
 - assigning to group tables 133
 - monotone sequencing in ~ 133
- dT parameter 95
- dynamic instantiation 112

E

- editor
 - ESDL ~ 113
- element 89
 - basic 144
 - graphical representation 143, 145
 - scalar 145
 - scope 99, 145
- Entry action 47
- entry action
 - in state machines 139
- enumeration 96
- equality operator 117
- ERCOS^{EK} 14, 20, 21

ESDL

- access block diagram 138
- basic elements 112
- description 111
- direct access methods 127
- feature list 111
- general features 111
- implementation cast 119
- instantiation 112
- Java syntax in ~ 111
- syntax 114

ESDL editor 113

Euler 180

- exit action 48
- in state machines 139

exp() 137

expression 114

external event 15

external source code 169

F

floor() 137

fmod() 137

for 123

G

Get/Set ports 147

getAt()

- array elements 128
- matrix elements 129
- table elements 130
- two-dimensional table elements 132

graphical representation

- element 145
- expression 145
- operators 151
- statement 154

group tables 133

- assigning distributions 133
- public interface 133

H

Heun 180

hierarchy 41

- of classes 31
- of modules 31
- of state machines 57

History

- state machine 44, 57

hybrid project 173

I

if...then statement 156

if...then...else operator 157

if...else 120

If...Then 156

If...Then...Else 157

implementation 103

- code generation 108
- composite types 105
- implementation cast 106
- scalar types 103
- transformation 109
- user defined types 105

implementation cast 97, 106–108, 149

ESDL 119

inheritance 112

instantiation 25

integration method

- Adams-Moulton 181
- Euler 180
- fixed step size 178
- Heun 180
- Mulstep 180
- Runge-Kutta 181
- variable step size 178

Integration step size 188

interface

- of a class 27
- of a component 27
- of a module 29

Interface Editor 113

interpolate() 130, 132, 134

interpolation

- linear 130
- rounded 130

interpolation mode

- ~ of tables 130
- linear 130, 132

interprocess communication 20

J

Java programming language

see programming languages

Junction 38
state machine 51

K

keywords
reserved ~ in ESDL 114
kind 96

L

length() 128
library functions
accessing ~ 136
limit() 137
linear interpolation 130, 132
Literal 146
literal 96, 116
local variable
C code 166
log 115
log() 137
log10() 137
logical operator 117, 151
loops
see *control flow*

M

mathematical functions
accessing library functions 136
primitive methods 116
MathFcn 136
matrix 92, 129, 146
access in ESDL 129
Get/Set Port 147
maximum size of ~ 129
public interface 129
Max operator 153
max() 116, 137
message 20, 94, 165
~ in processes 135
accessing ~ in ESDL 135
methods
arguments 113
editing method bodies 112
header 112
interface 112
method calls 124

naming conventions 112
nesting method calls 124
overloading 113
precedence of method calls 117
primitive methods 116
private 126
public 126
return value 113, 124

Min operator 153

min() 116, 137

model type

continuous 90
logical 90
signed discrete 90
unsigned discrete 90

module 19, 24

hierarchical structure 31

interface 29

vs. class 24

Mulstep 180

multi-tasking 15

Multiplex operator 152

MUX 118, 152

see *conditional operator*

N

Negation operator 154

non-preemptable scheduling 17

Not 151

O

object

access control 126

object reference

this in method calls 126

object-oriented concepts 112

Of state machines 41

one-dimensional table 92, 129

C code interpolation 167

interpolation mode 130

linear interpolation 130

maximum size 130

public interface 130

operating system

real time ~ 13

- operator 117–118
 - Abs 154
 - arithmetic 151
 - arithmetic ~ 117
 - associativity of ~ 118
 - Between 154
 - case 153
 - comparison 151
 - comparison and equality ~ 117
 - conditional ~ 118, 152
 - logical 117, 151
 - Max 153
 - Min 153
 - Multiplex 152
 - Negation 154
 - order of evaluation 150
 - precedence levels 118
 - shorthand assignment ~ 118
 - unary ~ 117

- OR 151
 - see logical operators
- overloading 113

P

- parameter 96
 - dependent ~ 99
 - see also arguments
 - virtual ~ 99
- pi() 137
- PMI 170
- pointers 111
- pow() 137
- precedence
 - ~ of operators 117–118
- pre-emptive scheduling 16
- priority
 - task 16
- private
 - see *accessing objects*
- process 15, 19
 - using messages 135
 - see also methods 112
- programming languages
 - C 161
 - C vs. ESDL 111, 141
 - Java vs. ESDL 112, 142
- Programming Model Interface 170

- project 13
 - hybrid 209
 - module 20
 - process 20
- public
 - see *accessing objects*

R

- real time operating system 13
- real-time
 - dT parameter 95
 - language construct 94
 - message 94
 - resource 95
- records
 - see data structures
- relational operator 117
- reserved keywords 114
- resource 95, 149
- return 125
- return value 113, 124
- reusing components 29
- rounded interpolation 130
- Runge-Kutta 181

S

- sch() 137
- scheduling 15
 - cooperative 15
 - non-preemptible 17
 - pre-emptive 16
- scope 99
- sdisc 115
- search() 130, 132, 134
- self
 - see this
- Semantics
 - hierarchical state machine 57
 - simple state machines 48
 - state machine with junction 51
 - state machines 47–71
- sequence call 155
- sequence number 155
- sequences
 - atomic ~ 141
- sequencing 159

- setAt()
 - array elements 128
 - matrix elements 129
- shift operators 111
- shorthand assignment operator 118
- sign() 137
- sin() 137
- sinh() 137
- software event 15
- specification
 - component 23
 - in C code 161
- sqrt() 137
- Start state 43
- State 34
 - entry action 47
 - exit action 48
 - static action 47
- State diagram 33
- State editor 140
- State machine 32–87
 - action 45
 - class 83
 - condition 45
 - ESDL in ~ 139
 - function 47
 - hierarchy 41, 57
 - history 44, 57
 - inlining 73
 - junction 38, 51
 - optimize (actions) 75, 76
 - optimize (conditions) 75, 76
 - optimize (hierarchical code generation) 81
 - optimize (junctions) 76
 - optimize (static action of hierarchy state) 77
 - optimized for code size 76
 - optimized for response time 74
 - optimized for runtime 75
 - outlining 73
 - semantics 47–71
 - start state 43
 - transition 33, 35
 - trigger 41
- statements 114
 - block statements 114

- static action 47
 - in state machines 139
- strings 112
- Switch 157
- switch...case...default 121
 - fall through 122
- syntax
 - ESDL 114
 - method calls 124
- system constant 97
- System Library
 - Bit Operators 215
 - Comparators 221
 - Control 242
 - Counter 223
 - Delay 228
 - Integrators 248
 - Lowpass 252
 - Memory 233
 - Miscellaneous 236
 - Nonlinears 239
 - Timer 223

T

- table 129–134
 - group ~ 133
 - interpolation mode 130
 - linear interpolation 130, 132
 - one-dimensional 129
 - two-dimensional 131
- Table Editor 128
- tan() 137
- tanh() 137
- task 15, 17
 - priority 16
- this 126
- timer 15
- transformation 109
- Transition 33, 35
 - action 48
 - in state machines 140
 - priority 35
- Transition action 48
- Transition editor 140
- Trigger 41

- two-dimensional table 92
 - C code interpolation 167
 - linear interpolation 132
 - maximum size 131
 - public interface 132
 - see also table
- type
 - basic 89, 90
 - composite 91
 - scalar 90
 - user defined 89, 100
- type casting
 - see conversion

U

- udisc 115
- unary operators 117

V

- variable
 - local 166
- variables 96
 - declaration of ~ 115
 - direct access methods 127
 - naming conventions 114
 - public and private ~ 126
 - reserved keywords 114
 - temporary ~ 98
 - virtual ~ 99

W

- while 123
- While loop 158

X

- xLength() 129

Y

- yLength() 129