CIP SYSTEM AG

Communicating
Interacting
Processes

CIP

CIP Tool 5

Help

IP Tool Version 5.10.00)

Revision March 2009

CIP System AG

Ritterquai 23

Postfach

CH-4502 Solothurn

Switzerland

E-Mail: cip@ciptool.ch

Internet: http://www.ciptool.ch

Contents

CIP Tool Help

Using Bookmarks for Navigation in this Help Manual

Choose *Window > Show Bookmarks*
to display bookmarks in the overview area.

If a triangle appears to the left of a bookmark,
click the triangle to show or hide subordinate bookmarks.

Bookmarks refer to descriptions of CIP modeling constructs. The nesting
of bookmarks reflects the compositional structure of CIP models.

To go to the text referred by a bookmark, click the bookmark text.

Operating Systems

CIP Tool is compatible with:

Windows NT, 2000, XP

⇨ CIP Tool is **not compatible** with: **Windows Vista**

Setting CIP Tool Preferences

You can customize several properties of CIP Tool using the **<Preferences>** command of the CIP browsers **<Options>** menu. The preferences are grouped in three main parts, each containing several options:

Look & Feel

The look options influence the appearance of CIP Tool:

<Look & Feel>
adapts the look for windows and menus to the selected platform.

<Gray Scale>
defines the background gray darkness of graphic editors.

Mouse options
determine the behavior of the mouse
(see also *CIP Browser*):

<Enable Window Button>
makes the right mouse button show window commands.

<Swap Object and Window Button>
If the window button is enabled, this option interchanges the default mouse button configuration.

Fonts

The fonts options let you determine the fonts and sizes to be used for different CIP Tool texts.

Utilities

The utilities options define general aspects of CIP Tool behavior:

<Backup spec files>
esisting specification files are renamed to .bak before a specification is saved.

<Restore editors>
graphic editors that are open when the system is saved are opened automatically when the saved system is opened again.

<Show tool tips>
enables explanatory tool tips for list and editor buttons.

<Show popup info>
enables information on graphic objects to appear when the cursor is placed within the object.

<Show comment popup>
enables comments on graphic objects to appear when the cursor is placed within the object.

<Warn with beep>
sounds a beep when an invalid action is performed.

<Show hints>
shows explanatory hints for invalid actions.

<Print extra page>
adds an extra page at the end of report and hardcopy.

<use postscript>
report and hardcopy are generated in postscript format.

Use the **Accept** button to save the settings leaving open the preferences editor.

Use the **Ok** button to save the settings and close the preferences editor.

Use the **Cancel** button to reject the changes and close the preferences editor.

The Mouse Has Two or Three Buttons!

The mouse behavior depends on the **Mouse Options** set in the Preferences Editor.

Meaning of the Buttons

The three mouse buttons are designated, according to their function, as the *Selection Button*, *Object Button* and *Window Button* respectively.

	Control area	Actions
Selection Button	Button, Checkbox	assign, mark, switch
	List element	select
	Graphic element	select, move, connect, delete, open (double click)
	Text edit field	select, position cursor
Object Button	Object list	Menu for selected list element
	Graphic element	Menu for displayed graphic element
	Edit field	Menu with edit functions
Window Button optionally enabled	Window	Menu with standard window functions

Mouse Implementations

In mice with only one or two physical buttons the mouse button must be used in combination with the *Alt* and the *Control* key:

Object Button = Button + *Alt* key

Window Button = Button + *Control* key

Constructing CIP Models

A CIP model is constructed by creating, composing and connecting modeling elements such as states, operations, processes or channels. All these modeling elements are called *CIP objects*.

Creating CIP Objects

A new CIP object is created with the **<new>** menu command of the corresponding CIP object list.

New CIP objects can be created also using the <copy> and <paste> commands of an object list.

Systems, clusters, processes and channels can be saved/exported and opened/imported as model files (linear form as ASCII text).

The editor framework of the tool reflects the compositional structure of CIP models:

Systems, clusters, processes and modes are created in the CIP browser. The editors for CIP objects contained in these components are accessed from the corresponding list menu of the CIP browser.

Remark:
The object list for a particular CIP object type appears in general in various editors as assignment list. The <new> command, however, is available in the basic object list only.

Associating CIP Objects

Associations among CIP objects are created by assignment. Thus most CIP object editors contain one or more lists of CIP objects which can be assigned to the edited object.

Creating and deleting associations

Creating an association:
- select the CIP object(s) to which another object shall be assigned
- select in an assignment list the CIP object to be assigned
- click the assignment button

Deleting an association:
- select the CIP object with the assignment to be removed
- activate the corresponding assignment list
  (in some cases the element to be deassigned must be selected also)
- click the deassignment button

Interconnecting Graphic CIP Objects

Connectors among graphically represented CIP objects are created or deleted with connection tools of the corresponding graphic editor.

Creating and deleting directed connections

Creating a connection:
press the selection button within the first element to be connected;
drag the cursor to the second element to be connected and release.

Deleting a connection:
press the selection button within the first element;
drag the cursor to the connected element and release the mouse button.

CIP Browser

A distinction is made between three different work areas in the tool. You can switch to another area by clicking the corresponding button underneath the Logo box.

Specifications

Complete CIP models are constructed by means of modelling editors accessible from the CIP specification browser. The browser hierarchy reflects the compositional structure of a CIP model.

Implementations

The implementation browser serves to generate C-code for the implementation of a CIP model. For this, the set of clusters is partitioned into concurrent CIP units. A collection of C-modules is created for each CIP unit.

Animations

The animations browser is used to generate C-code for visual and textual animations of CIP units. An animation represents an implementation of a CIP unit, in which keyboard, mouse and screen replace the connection to real environment.

SYSTEMS

CIP Systems are created or opened from the CIP browsers **<FILE>** menu.

SYSTEM

consists of	SYSTEM INCLUDE, CONSTANTS, STANDARD TYPES, CIP RECORDS, USER TYPES, INDEX TYPES, CHANNELS, COMMUNICATION NET, CLUSTERS

A CIP system is composed of a set of asynchronously cooperating clusters, each consisting of a number of synchronously cooperating state machines termed processes. Formally a cluster represents a state machine also, but one with a multi-dimensional state space. The processes of a cluster interact by means of synchronous many-casts of internal events called pulses. All processes of a CIP system can communicate asynchronously with each other and with the environment by means of channels. Multiple instances of a process are specified as process arrays.

A cluster is activated by a transmitted channel message which leads to a state transition of the receiving process. By emitting a pulse, the receiving process can activate further processes of the cluster, which can in turn activate other processes by pulses. The chain reaction resulting from pulse transmission is not interruptible and defines a single state transition of the entire cluster. Activated processes can also write messages to their output channels.

SYSTEM INCLUDE

is part of a	SYSTEM	CIP Browser -> SYSTEMS Menu -> <INCLUDE>
consists of	C or C++ source code statements	

The system include text is inserted at code generation time in the *CIP SHELL* header files of the system.

Version Control

is part of a	SYSTEM	CIP Browser -> SYSTEMS Menu -> <Version Control>
consists of	version control keywords and values text	

The version control text is inserted near the top of the system specification file (.cip) and can be modified by the version control system at check in time.

It also is inserted in the header comment of each generated code file.

CONSTANT

is part of a	SYSTEM	CIP Browser -> SYSTEMS Menu -> <CONSTANTS>
consists of	STANDARD TYPE VALUE	
is assigned to	INDEX TYPES	
is used in	CONDITIONS, OPERATIONS, DELAYS, INQUIRIES, FUNCTIONS, SELECTORS	

A constant defines a value of a STANDARD TYPE.

Integer constants can be assigned to *INDEX TYPE*S to specify their range.

CONSTANT Editor

The constant type is specified by an assigned STANDARD TYPE. The constant value is defined with the line editor underneath the CONSTANTS List.

Use of constant names in C-code constructs

A constant is implemented in the generated code as #define ConstantName VALUE.

Constant names can be used in operations, delays, conditions, functions inquiries, selectors.

(see also *CIP MACROS*)

STANDARD TYPE

is part of a	SYSTEM	CIP Browser -> SYSTEMS Menu -> <STANDARD TYPES>
consists of	elementary ANSI C data type	
is assigned to	VARIABLES, CIP RECORD FIELDS	

Standard types represent predefined ANSI C type definitions:

BOOLEAN defines `int`

INTEGERdefines `int`

FLOATdefines `float`

STRINGdefines `char[81]`

Remark: If code is generated for textual animations or for implementations with the trace code option set, the produced code supports displaying of standard type values.

CIP RECORD

is part of a	SYSTEM	CIP Browser -> SYSTEMS Menu -> <CIP RECORDS>
consists of	FIELDS with assigned STANDARD TYPE or USER TYPE	
is assigned to	VARIABLES, MESSAGES, INPULSES, OUTPULSES	

A CIP record is a composed data type (`struct`) consisting of one or more fields with assigned standard type or user type.

Remark: If code is generated for textual animations or for implementations with the trace code option set, the produced code supports displaying of standard type values.

CIP RECORD Editor

The CIP RECORD Editor is a browser which displays the FIELDS List for the CIP RECORD selected. You can create the FIELDS of the CIP RECORD in this list and assign to them types from the switchable *STANDARD TYPE*S / *USER TYPE*S List.

Use of field names in C-code constructs

The names of CIP record fields are used as component identifiers of a correspondingly generated C structure. CIP record fields are accessed in user code constructs correspondingly by field names:

Accessing a CIP record field *myField*

of an inpulse or an inport message: `IN.myField`.

of an outpulse: `OUTPULSE.myField`.

of an outport message: `OUTMSG.myField`.

Accessing a CIP record field *myField* of a variable myVar:

write and read access by	`SELF.myVar.myField`
read only access by	`STATUS.myVar.myField`

(see also *CIP MACROS*)

USER TYPE

is part of a	SYSTEM	CIP Browser -> SYSTEMS Menu -> <USER TYPES>
consists of	ANSI C type or C++ class definition	
is assigned to	VARIABLES, MESSAGES, INPULSES, OUTPULSES, CIP RECORD FIELDS	

A user type is defined by an ANSI C typedef construct or by a C++ class definition.

USER TYPE Editor

When a new USER TYPE is created, the text editor already contains the word `typedef` and the name of the data type.

The name of the edited data type must be the same as the name of the USER TYPE: When the USER TYPE is renamed, the name of the data type in the typedef-construct must be post-edited.

INDEX TYPE

is part of a	System	CIP Browser -> SYSTEMS Menu -><INDEX TYPES>
assigned	INTEGER CONSTANT	
is assigned to	PROCESS, OUTPORT and CHANNEL MULTIPLICITIES	

An index type represents an elementary multiplicity used for the specification of process, outport and channel array dimensions.

INDEX TYPE Editor

The RANGE of an INDEX TYPE is defined by assigning an integer constant from the CONSTANTS List.

A constant with value N defines the range 0,1, ..., N-1.

Use of index type names in C-code constructs

Index type names are used in operations, delays, conditions, functions and selectors to access the index values of the active process instance:

`ID.indexTypeName` contains the index value of the corresponding array dimension.

(see also *CIP MACROS*)

CHANNEL

part of a	SYSTEM	CIP Browser -> SYSTEMS Menu -> <CHANNELS>	
boolean attribute	CONTROLLED		
consists of	MESSAGES, MULTIPLICITY		
is node of	COMMUNICATION NET	connected to	PROCESSES

Channels model unidirectional asynchronous communication connections which retain the sequential order of transmitted messages. Asynchronous communication in a CIP model means that the write and the read action of a message transmission takes place in different cluster transitions.

Ordinary channels transmit their messages regardless of the current state of the reader process. *Controlled* channels in turn deliver their messages only when the reader is in a state expecting messages from that channel.

Every channel can be represented as node in each of the graphical communication subnets.

CHANNEL Editor

The CHANNEL Editor is implemented as CHANNEL CATEGORY Browser. CHANNEL CATEGORIES are created by the user to classify informally the channels of a CIP model. Typical categories are EventChannels (sources), ActionChannels (sinks) and SystemChannels (internal).

If a channel of a category is selected, the MESSAGES List of the channel is displayed. In this list you can create, rename and delete messages. Furthermore, data types can be assigned from a switchable *CIP RECORD*S / *USER TYPE*S List.

With the <MULTIPLICITY> command you open an editor for specifying a channel array consisting of several channel instances. The MULTIPLICITY Editor allows you to directly specify an INDEX TUPLE defining the multiplicity of the channel by assigning *INDEX TYPE*S.

With the <CONTROL> command you open a prompter with a check box, which allows you to specify the channel as controlled channel. Ordinary channels transmit their messages regardless of the current state of the reader process. Controlled channels in turn deliver their messages only when the reader is in a state expecting a message from that channel. Controlled channels are marked in the CHANNELS List with c. In the graphic communication subnets controlled channels are marked in gray.

The code generator supports the automatic implementation of internal ordinary channels. Controlled channels must always be implemented by the user.

CHANNEL MESSAGE

is part of a	CHANNEL	CHANNEL Editor -> MESSAGES List
optionally assigned	CIP RECORD or USER TYPE	

A channel message is an elementary transmission unit of a channel. Messages without data type represent pure signals. Messages with assigned *CIP RECORD* or *USER TYPE* carry data.

CHANNEL MULTIPLICITY

is part of a	CHANNEL	CHANNEL Editor -> CHANNELS Menu -> <MULTIPLICITY>

optionally assigned INDEX TYPES

The multiplicity of a channel array is specifed by an INDEX TUPLE consisting of one ore more assigned *INDEX TYPE*S. An INDEX TYPE can be used no more than once per channel.

Connection conditions (checked automatically before code generation):

The reader array must have the same multiplicity as the channel array. Every index type of the channel array must be an index type of the sender array or of a corresponding outport array.

Communication at instance level is one-to-one with respect to common index types, otherwise it is many-to-one.

COMMUNICATION NET

is part of a	SYSTEM
union of	COMMUNICATION SUBNETS

The communication net of a CIP model is specified as union of several graphically modelled subnets. The complete communication net can be inspected in a channel connection browser which is opened with <list channel connections> from the SYSTEMS List menu of the CIP Browser. The part of the communication net that is not visile in any of the subnets can be seen in a browser opened with <invisible communication> from the SYSTEMS menu.

Processes communicate asynchronously by means of channels attached to process ports. Channel communication is the only means to transmit information from one cluster to an other. Source and sink channels model the interface to the system environment.

Channels model an active communication medium which retains the sequential order of transmitted messages. Asynchronous communication in a CIP model means that the write and the read action of a message transmission takes place in different cluster transitions. Processes represent receptive behavioral entities which must accept delivered messages at any time. The buffer size of a channel is not specified in the CIP model.

If for a channel the CONTROLLED attribute is set, the channel models a passive communication medium, i. e. a message is released when the reader process awaits a message of the attached inport. The CONTROLLED attribute should be set in special cases only (c.f. order queuing) because the throughput of the passive communication model is dependent of the modelled system behavior.

COMMUNICATION SUBNET

is part of a	SYSTEM	CIP Browser -> SYSTEMS Menu -> <COMMUNICATION> ▸ COMMUNICATION NETS
consists of	graphical communication connections	

Communication subnets represent graphical views of the communication net model of a system.

COMMUNICATION SUBNET Editor

The COMMUNICATION SUBNET Editor allows you to graphically model the communication structure of CIP system by individual subnets. The subnets may overlap; that is, they may contain common parts. The resulting redundancy in the graphic representation is automatically held consistent by the tool: If a process and a channel are represented in a communication subnet, an existing communication connection is also visible.

Tool Bar

Representing processes as graphic nodes:
select element in *PROCESS*ES List; place cursor; click.

Representing channels as graphic nodes:
select element in *CHANNEL*S List; place cursor; click; place label; click.

Erasing graphic nodes:
click on node to be erased.
Erasing a graphic node has no effect on the system's communication net model.

Interconnecting channels and processes:
press the selection button within the first element to be connected;
drag the cursor to the second element to be connected and release.

Disconnecting channels and processes:
press the selection button within the first element;
drag the cursor to the connected element and release.

Selecting or moving:
click or drag.
multiple selection: shift-key + click or drag a selection window.
double click to open an editor for the selected element

Erasing Channels and Processes

If a graphic channel or a process symbol of a communication subnet is erased, the communication connections concerned are no longer represented. Erasing a process or a channel in a communication subnet only means that erased node and its connections are no longer visible in that communication subnet. Erasing even a communication subnet completely does not change the specification of the logical communication net model. Channel-process connections not visible in any graphical subnet are collected in an invisible connection list, which is opened with <list invisible connections> from the SYSTEMS List menu of the CIP Browser.

Connection conditions

When a port of a process is connected with a channel, the messages of the port are identified with the corresponding messages of the channel. The set of messages of the channel is called channel protocol. The graphic connection function permits only valid connections.

Thus when you connect a port and a channel, the messages of the port must correspond to the messages of the channel. Correspondence means same name and same type. If the unconnected channel contains more messages then the port, the set of messages of the port is extended automatically. Vice versa, if the unconnected port contains more messages than the channel, the channel protocol is extended.

Changing the set of messages of a channel or of a port of a connection or changing the data type of a message means changing the common channel protocol. The sets of messages of all parts of the connection are automatically held consistent.

Double clicking a channel opens the channel editor and double clicking a process opens a mode graph of the process.

implementation attributes

is part of a	SYSTEM	CIP Browser -> SYSTEMS Menu -> <implementation attributes>
consists of	naming option	
is assigned to	CHANNEL, MESSAGE	

Implementation attributes are used to make message names globally unique for a system. This is achieved by adding a prefix to the message names. The names are used as enumeration labels for the messages.

use default prefix	The default prefix is used: *C#_aMessage* with # representing the enumeration value of the channel.
channel name prefix	The name of the channel is used as prefix to the message name (*aChannel_aMessage*).
channel tag prefix	A tag can be defined for each channel. This tag is used as prefic to the message name (*mytag_aMessage*). If no tag is defined for a channel, the default prefix is used.
user defined	A user defined name is used as enumeration label for the message. If no user name is defined, the specification name of the message is used.

CLUSTERS

A system consists of a set of asynchronously cooperating clusters.

CLUSTER

is part of a	SYSTEM	CIP Browser -> CLUSTERS List
consists of	CLUSTER INCLUDE, PULSE CAST NET, PULSE TRANSLATIONS, INSPECTION NET, MODE CONTROL NET, INITIALIZATION, PROCESSES	

The clusters of a CIP model represent concurrent functional blocks. A cluster is a state machine consisting of several synchronously cooperating processes. The processes of a cluster can interact by means of synchronous many-casts of internal events termed pulses. Processes can communicate asynchronously by means of channels with the environment and with all processes of a CIP model.

A cluster is always activated by a channel message which leads to a state transition of the receiving process. By emitting a pulse, the receiving process can activate further processes of the cluster, which can in turn activate other processes by pulses. The chain reaction resulting from pulse transmission is not interruptible and defines a single state transition of the entire cluster. Activated processes can also write messages to their output channels.

Singular clusters as well as cluster groups can be transformed automatically into concurrently executable software components (see *CIP UNIT*S).

CLUSTER INCLUDE

is part of a	CLUSTER	CIP Browser -> CLUSTERS Menu -> <INCLUDE>
consists of	C or C++ source code statements	

The cluster include text is inserted at code generation time in the code files of the created process modules of the cluster.

PULSE CAST NET

is part of a	CLUSTER	CIP Browser -> CLUSTERS Menu -> <PULSE CAST>
union of	PULSE CAST SUBNETS	

The pulse cast net is specified as a union of several graphically modelled subnets.

The processes of a cluster interact synchronously by means of multi-cast pulses. Pulses represent internally transmitted events.

PULSE CAST SUBNET

is part of a	CLUSTER	CIP Browser -> CLUSTERS Menu -> <PULSE CAST> ▸ <PULSE CAST NETS>
consists of	processes, graphical pulse cast connections	

A pulse cast subnet is an architectural model defining the connections for pulse transmission among the processes of a cluster.

PULSE CAST SUBNET Editor

The PULSE CAST SUBNET editor allows you to define the pulse cast structure of a cluster by means of directed connections among processes with a round pulse cast connector in between. The subnets may overlap; that is, they may contain common parts. The resulting redundancy in the graphic representation is automatically held consistent by the tool: If both, a pulse cast sender and a receiver process are represented in a pulse cast subnet, existing pulse cast connectors are also visible.

Every pulse cast connector has an associated pulse translation which relates outpulses of the sender to inpulses of the receiver process. To open the pulse translation editor double click the pulse cast connector.

Tool Bar

Representing processes as graphic nodes:
select element in *PROCESS*ES List; place cursor; click.

Creating pulse cast connections:
press the selection button within the intended sender process;
drag the cursor to the intended receiver process and release.

Removing pulse cast connections:
click on the connector circle to be deleted.

Erasing graphic processes:
click on node to be erased.
Erasing a graphic node has no effect on the clusters pulse cast net model.

Selecting or moving:
click or drag.
multiple selection: shift-key + click or drag a selection window.
double click to open an editor for the selected element

CAST ORDER

is part of a	CLUSTER	CIP Browser -> CLUSTERS Menu -> <CAST ORDER>
refers to a	SENDER	
defines	RECEIVER ORDER	

For models with multicast pulse transmissions the structure of the pulse cast net does not sufficiently restrict the potential pulse transmission chains, as non-deterministic process activations are in general possible. To ensure deterministic pulse propagation, pulse cast interaction is specified as sequential multicast. The outgoing pulse cast connections of each process are therefore defined as a totally ordered set. If a process emits a pulse, the receivers determined by the pulse translation function are triggered sequentially in the specified cast order to cause subsequent pulse cast chains.

CAST ORDER Editor

For a process selected in the SENDERS List the receivers defined in the pulse cast net are displayed in the RECEIVERS List. The list order from top to bottom defines the activation order of the receivers when the sender emits an outpulse.

This cast order can be changed by selecting a receiver and by clicking the up- or down-button beneath the RECEIVERS List.

PULSE TRANSLATION

is part of a	PULSE CAST CONNECTION	CLUSTERS Menu -> <PULSE TRANSLATION>

refers to	SENDER OUTPULSE
optionally assigned	RECEIVER INPULSE, SENDER SELECTOR

The pulse cast net models the pulse flow structure of the cluster. The transmission of pulses between a sender and a receiver must be specified by associating the sent outpulses with the inpulses to be received. Pulse translations represent the glue of an pulse cast connection.

PULSE TRANSLATION Editor

For a selected SENDER its set of *OUTPULSE*S is displayed in the TRANSLATIONS List. In the RECEIVERS List the possible receiver processes appear. Selecting a RECEIVER opens in the right-hand section of the editor the corresponding *INPULSE*S List.

The collection of sender/receiver pairs is defined by the pulse cast connections of the pulse cast net. For every sender/receiver pair a translation function maps outpulses of the sender partially into inpulses of the receiver. A singular pulse translation is defined by assigning a receiver inpulse to a sender outpulse. A receiver's inpulse can be assigned to several outpulses of a receiver. In general, a sender's outpulse is not translated to an inpulse of every receiver. At run time, untranslated outpulses are not transmitted.

A pulse transmission of a sender to a process array may cause several instances of the receiver to be activated (many-cast). Transmissions of this kind can be restricted at instance level by PULSE SELECTOR Assignment (see *SELECTOR*).

Interaction Trees

generated by the tool CIP Browser -> CLUSTERS Menu -> <Interaction Trees>

The specification of process reactions and pulse cast interactions by means of state transition diagrams and pulse translation functions defines a deterministic cluster model, from which all potential pulse cast sequences can be deduced. The construction of models with cyclic pulse cast sequences is prevented by the tool. (An pulse cast sequence is termed cyclic, if the same process is activated more than once by the same input.)

Interaction Tree Viewer

All potential pulse cast sequences of a CIP model in construction can be viewed graphically as interaction trees. To view a particular interaction tree select a PROCESS and a PORT in the process interface browser at the left hand side of the viewer. Then select an inport message, an inpulse, an outpulse or an extension trigger in the displayed TRIGGERS List. The maximal pulse cast sequences caused by the selected trigger are displayed as graphic interaction tree.

Pulse cast propagation takes place as left traverse of the displayed tree structure. If an outpulse of a process is transmitted to several receivers, the multicast is performed sequentially corresponding to the specified CAST ORDER.

If a process can react to a particular inpulse by emitting one of several outpulses, the pulse cast propagation forks into alternative interaction branches. At run time, the choice of the branch is determined by the executed state transition.

An interaction tree shows the maximal pulse cast sequences that can be caused by its root trigger. The effectively executed pulse cast sequence at run time depends on the active modes, current states and variables of the cluster processes.

Cascades

generated by the tool	CIP Browser -> CLUSTERS Menu -> <Cascades>

The maximal control flow caused by an externally triggered process is viewed as process cascade. A cascade represents an envelope of all pulse cast sequences caused by the individual external inputs of the cascade root process.

Cascade Viewer

To view a process cascade select a process in the PROCESSES List at the left hand side of the viewer. The cascade is displayed as control flow tree which is executed as left traverse of the tree structure.

Cascades show maximal activation sequences of processes. At run time, the set of activated processes depends on the external input, on the modes, states and variables of the cluster processes.

INSPECTION NET

is part of a	CLUSTER	CIP Browser -> CLUSTERS Menu -> <INSPECTION>
union of	INSPECTION SUBNETS	

The inspection net is specified as a union of several graphically modelled subnets.

A process allows state inspection by connecting to a *GATE* of an inspecting process within the same cluster. A gate represents a boolean function returning wether the inspected process is actually in a particular state or in a state contained in a specified set. The function is defined by means of an associated *TRUTH TABLE*. At runtime, the required information is obtained by evaluating the gate.

A gate is usually assigned as guard in a switch. Within a process, you can logically combine gates in conditions using the gate names like macro calls: e.g.
`aGate1Name() || aGate2Name()`

A process can also inspect the state vectors of the other cluster processes. The required information is obtained by calling process inquiries (process access functions offered by the inspected process).

Calling inquiries is usable in operations, delays, conditions, functions, inquiries and selectors.

INSPECTION SUBNET

is part of a	CLUSTER	CIP Browser -> CLUSTERS Menu -> <INSPECTION> ▸ <INSPECTION NETS>
consists of	processes, graphical state inspection and state vector inspection connections	

The inspection subnet is an architectural model defining the connections for state inspection and state vector inspection among the processes of a cluster.

INSPECTION SUBNET Editor

The INSPECTION SUBNET editor allows you to define the inspection structure of a cluster by means of directed inspection connections among processes with a rhombic inspection connector in between. The subnets may overlap; that is, they may contain common parts. The resulting redundancy in the graphic representation is automatically held consistent by the tool: If both, an inspector and an inspected process are represented in an inspection subnet, existing inspection connectors are also visible.

State Inspection

State inspections relate inspcted processes to a gate of an inspecting process. They are represented by white rhombic connectors from the inspected to the inspecting process. Up to three inspected processes can connect to one gate. The connector direction indicates the data flow.
To open the truth table associated to a gate double click the state inspection connector.

State Vector Inspection

State vector inspections enable the inspecting process to call inquiries of the inspected process. They are represented by gray rhombic connectors from the inspected to the inspecting process. The connector direction indicates the data flow.
To open the inquiries editor of the inspected process double click the state vector inspection connector.

Tool Bar

Representing processes as graphic nodes:
   select element in *PROCESS*ES List; place cursor; click.

Creating state inspection connections:
   press the selection button within the intended inspected process;
   drag the cursor to the intended inspecting process and release;
   create or choose a gate of the inspecting process to complete connection to the inspected process.

Creating state vector inspection connections:
   press the selection button within the intended inspected process;
   drag the cursor to the intended inspecting process and release.

Removing inspection connections:
   click on the rhombic connector;
   for multiple state inspection connection select the process to be exempt from inspection;

Erasing graphic processes:
   click on node to be erased.
   Erasing a graphic node has no effect on the clusters inspection net model.

Selecting or moving:
   click or drag.
   multiple selection: shift-key + click or drag a selection window.
   double click to open an editor for the selected element

TRUTH TABLE

is part of a	state inspection CONNECTOR	inspection editor -> state inspection connector menu>
consists of	STATE-Truth associations	

The inspection truth table of a gate defines the value to be returned for each state combination of the inspected processes.

TRUTH TABLE Editor

The layout of the editor depends on the number of inspected processes connected.

One Inspection:

> A one dimensional truth table is used to define for every state of the inspected process a corresponding boolean value. The table appears as row or column of state fields.

Two Inspections:

> A two dimensional truth table is used to define for every pair of inspected process states the corresponding boolean value.

Three Inspections:

> The truth table is a three dimensional structure. Three distinct state browsers give three different views on the truth table:

For every browser, one of the three inspected processes is fixed as guiding process. The truth table is correspondingly viewed as a browsable collection of two dimensional tables, enumerated by the states of the guiding process. For a selected guiding process state, the browser allows you to edit the corresponding two dimensional part of the truth table.

Adding an inspected process

> When a further process is to be inspected, the dimension of the corresponding truth table increments by one. The old truth table can be retained for one state of the new inspected process. You are prompted to choose this state when you connect graphically the new process to the state inspection connector.

Removing an inspected process

> When a process is removed from the state inspection, the dimension of the corresponding truth table decrements by one. A part of the old mode setting table can be retained, namely the sub table associated to a fixed state of the process to be removed. You are prompted to choose this state when you disconnect graphically the inspected process from the state inspection connector.

Tool Bar

● True ○ False ○ Clear	Preset a boolean value: select one of the values; the chosen value is filled in a table field by clicking in the field.
Invert all	Invert all defined table elements: click on the button invert the value of the table fields containing a boolean value.
All True	Change all table elements: click on the button to set all table elements to the preset value;
Complete	Complete truth table: click on the button to complete the table with the preset value.

MODE CONTROL NET

is part of a	CLUSTER	CIP Browser -> CLUSTERS Menu -> <MODE CONTROL>
union of	MODE CONTROL SUBNETS	

The mode control net is specified as a union of several graphically modelled subnets.

The mode changes of a *slave* process can be induced by up to three processes designated as *master*. The active mode of a slave process is determined by the current states of its masters. Any process defining at least one state can be a master.
A Process is automatically marked slave as soon as it contains more than one mode.

MODE CONTROL SUBNET

is part of a	CLUSTER	CIP Browser -> CLUSTERS Menu -> <MODE CONTROL> ▸ <MODE CONTROL NETS>
union of	processes, graphical mode control connections	

MODE CONTROL SUBNET Editor

The mode control relation of a cluster is specified by an acyclic process graph. Mode control connections are represented by triangles which are connected at the bottom angle to a slave and at the top side to one or more masters. The subnets may overlap; that is, they may contain common parts. The resulting redundancy in the graphic representation is automatically held consistent by the tool: If both, a slave and a master process are represented in a mode control subnet, existing mode control connections are also visible.

Tool Bar

Representing processes as graphic nodes:
select element in *PROCESS*ES List; place cursor; click.

Creating mode control connections:
press the selection button within the intended master process;
drag the cursor to the intended slave process and release.
If the slave is already connected to a master, a prompter asks you to select the state of the new master, for which the already specified mode setting has to be retained.

Removing mode control connections:
click the selection button within the mode control connector triangle.
If several masters are connected to the slave, a first prompter lets you choose the master process to be disconnected and a second prompter asks you to select the state of the disconnected master from which an already specified mode setting shall be retained for the remaining mode control ralation.

Erasing graphic processes:
click on node to be erased.
Erasing a graphic node has no effect on the clusters mode control net model.

Selecting or moving:
click or drag.
multiple selection: shift-key + click or drag a selection window.
double click to open editor for selected element

MODE CONTROL connector

The <MODE SETTING> menu command of the graphic mode control connector opens the MODE SETTING Editor where you specify in a table which master state enables which slave mode.

MODE SETTING TABLE

is part of a	MODE CONTROL CONNECTOR	mode control Editor -> connector menu>
consists of	MODE-STATE associations	

The mode setting table of a slave process defines how the active mode depends on the current master states.

MODE SETTING Editor

To assign a mode to a field of the table, select a mode (a drag cursor appears) and click in the table filed.

To remove an assigend mode from a table field, deselect all modes in the modes list (a cross cursor appears) and clich in the table field

The layout of the editor depends on the number of masters of a mode control relationship.

One Master:

A one dimensional mode setting table is used to define for every master state the corresponding active mode of the slave. The table appears as row or column of state fields.

Two Masters:

A two dimensional mode setting table is used to define for every pair of master states the corresponding active mode of the slave.

Three Masters:

The mode setting table is a three dimensional structure. Three distinct state browsers give three different views on the mode setting structure:

For every browser, one of the three masters is fixed as guiding master. The mode setting structure is correspondingly viewed as a browsable collection of two dimensional tables, enumerated by the states of the guiding master. For a selected guiding master state, the browser allows you to edit the corresponding two dimensional part of the mode setting structure.

Adding a MASTER

When a further master process is added to a mode control association, the dimension of the corresponding mode setting table increments by one. The old mode setting table can be retained for one state of the new master. You are prompted for this state when you connect graphically the new master to the mode control connector.

Removing a MASTER

When a master process is removed from a mode control association, the dimension of the corresponding mode setting table decrements by one. A part of the old mode setting table can be retained, namely the sub table associated to a fixed state of the master to be removed. You are prompted for this state when you disconnect graphically the master from the mode control connector.

INITIALIZATION

is part of a	CLUSTER	CIP Browser -> CLUSTERS Menu -> <INITIALIZATION>
consists of	initialization of states and process variables	

The initialization of a cluster is defined by the initial states, by the initial modes and by the initial variable values of all processes of the cluster.

The initial state of a process can be graphically specified in the *MODE* Editor. Process variables are initialized in the cluster's INITIALIZATION Editor.

INITIALIZATION Editor

The initial mode and state are defined by selection and execution of the list menu command <set>. The initial mode and state are marked with i. With <clear> the current initialization is canceled.

If a process is slave of one or more master processes, the initial slave mode, marked with x, is determined by the initial states of the masters.

With the menu command <initialize VARIABLES> of the PROCESSES List you open the initialization editor for process *VARIABLE*S. All variables are initialized to zero by default.

You can enter a value or the name of a *CONSTANT* for variables with *STANDARD TYPE* with the line editor underneath the VARIABLES List. Initialized variables are marked with i.

For *USER TYPE* process variables the <INIT OPERATION assignment> command opens an assignment editor for Initialization operations. These operations are created and edited in the INIT OPERATION Editor, opened with the <INIT OPERATIONS> command of the PROCESSES List menu.

For initialization of *CIP RECORD* process variables, a similar initialization editor for the CIP RECORD fields opens up when you enter the <initialize FIELDS> command.

INIT OPERATION

is part of a	PROCESS	INITIALIZATION Editor -> VARIABLES Menu
consists of	an ANSI C compound statement	

An initialization operation is an ANSI C compound statement used for initializing user type process variables.

Initialization operations are assigned to *USER TYPE* variables in the INIT OPERATION Assignment Editor, which is opened with <INIT OPERATION assignment> from the VARIABLE INITIALIZATION Editor.

A variable to be initialized is accessed with the CIP MACRO *VARIABLE*.

Example: `VARIABLE = 7.914;`

initializes a variable with of user type `typedef float myType;`

Initialization operations are also used for initializing fields of *CIP RECORD* process variables, when the assigned CIP RECORD contains *USER TYPE* fields. The field to be initialized is accessed by VARIABLE.fieldName.

PROCESSES

A cluster consists of a set of synchronously cooperating processes, specified as extended finite state machines.

PROCESS

is part of a	CLUSTER		CIP Browser -> PROCESSES List
consists of	PROCESS INCLUDE, STATES, VARIABLES, INPORTS, OUTPORTS, INPULSES, OUTPULSES, CONDITIONS, OPERATIONS, FUNCTIONS, GATES, INQUIRIES, MULTIPLICITY, SELECTORS, EXTENSIONS, DELAYS, MODES		
is node of	COMMUNICATION NET	connected to	CHANNELS
is node of	PULSE CAST NET	connected to	PROCESSES
is node of	INSPECTION NET	connected to	PROCESSES
is node of	MODE CONTROL NET	connected to	PROCESSES
extensions	TIMER, CHAIN, AUTO		

Processes represent reactive objects. By means of state transition structures and operations executed within transitions, functionality can be specified on two different levels of abstraction.

Pure state machines are extended by static process variables, data types for input and output elements, operations and conditions. All these extensions are formulated in the C/C++ programming language. From the high level modeling point of view operations and conditions represent primitives, which are automatically inserted inline in the generated CIP model code.

Operations executed in state transitions are used to access input and output data fields and to update the local process variables. Conditions serve to render non-deterministic state machines deterministic. Such conditions can depend on the input data and on the values of the local process variables, but also, by state inspection, on the states and variables of other cluster processes.

The communication interface of a process is defined by one or more inports and outports. A port is specified by the set of messages to be received or sent. Each inport and outport is connected in the communication net to an incoming or outgoing channel respectively.

Pulse cast inputs and outputs are defined by two distinct sets of inpulses and outpulses.

The full behavior of a process is specified by one or more state transition structures termed *mode*. Every mode of a process models a specific reactive behavior.

If a process has more than one mode, the active mode at run time is determined by the current state of one or more master processes. The masters of a slave are defined in the *MODE CONTROL NET* Editor.

Replicated processes are modelled as multidimensional process arrays. The multiplicities of the singular array dimensions are defined by abstract index types. Using common index types for different process arrays allows modeling of finite relations among process arrays, usually expressed by means of entity-relationship diagrams.

Process timers and feedback-mechanisms are supported by corresponding high level extensions. High level process extensions represent modeling sugar, i. e. the by the tool supported extensions could as well be specified by means of the basic CIP modeling constructs.

Processes appear automatically in the *PULSE CAST SUBNET* and the *MODE CONTROL NET* of the cluster as graphic nodes. The graphical representation of processes in the *COMMUNICATION SUBNET*S is specified by the user.

Use of process names in C-code constructs

Process names are used in inquiry calls:

`MyProcess.anInq()` calls the inquiry anInq of the process *MyProcess*.

PROCESS INCLUDE

is part of a	PROCESS	CIP Browser -> PROCESSES Menu -> <INCLUDE>
consists of	C or C++ source code statements	

The process include text is inserted at code generation time in the code file of the created process module.

STATE

is part of a	PROCESS	CIP Browser -> PROCESSES Menu -> <STATES>
boolean attribute	INITIAL	
is node of	MODE GRAPHS	connected toTRANSITIONS
is associated to	MODES of SLAVE PROCESSES	

A CIP process is an event driven extended finite state machine. State changes are triggered by messages and inpulses. A process is active only in state transitions.

States can either be created in the STATES List of the process or graphically in one of its *MODE* Editors.

Use of state names in C-code constructs

State names appear in the generated code as enumeration labels. They are used in conditions, functions and inquiries to address state values.

Example: `(STATUS.STATE == aState)` evaluates true if the current state is *aState*.

(see also *CIP MACROS*)

VARIABLE

is part of a	PROCESS	CIP Browser -> PROCESSES Menu -> <VARIABLES>
assigned	STANDARD TYPE or CIP RECORD or USER TYPE	

The memory of a pure state machine consists of the discrete state only. The process memory can be extended by means of local static variables. Process variables are read and updated by means of operations assigned to state transitions. In the VARIBLE Editor you must assign a *STANDARD TYPE*, a *CIP RECORD* or a *USER TYPE* to each variable.

Use of variable names in C-code constructs

Variables names are used in the generated code as component identifiers of the state vector structure of a process. State vector variables are accessed in user code constructs correspondingly by variable names:

Operations and delays:

Read and write access to a variable *myVar*:`SELF.myVar`.

Conditions, functions, inquiries and selectors:

Read access to a variable *myVar*: `STATUS.myVar`.

(see also *CIP MACROS*)

INTERFACE

The process interface consists on hand of inpulses and outpulses used to interact synchronously with other processes of the cluster, on the other hand of inport and outport messages allowing to communicate asynchronously with all processes of the system.

INPORT

is part of a	PROCESS	CIP Browser -> PROCESSES List -> <INTERFACE>
consists of	MESSAGES	
is connected to	CHANNELS	

An inport of a process defines a set of messages that can be received. Every inport of a process must be connected to an incoming channel (see *COMMUNICATION SUBNET*).

INPORT Editor

The INPORT Editor allows you to create inports with messages to which *CIP RECORD*S and *USER TYPE*S can be assigned.

INPORT MESSAGE

is part of a	OUTPORT	INPORT Editor -> MESSAGES List
optionally assigned	CIP RECORD or USER TYPE	
is assigned to	TRANSITIONS	

Inport messages without data type act as state transition triggers only. Messages with an assigned CIP record or user type carry data, accessed by means of operations executed in the triggered state transition.

INPULSE

is part of a	PROCESS	CIP Browser -> PROCESSES List -> <INTERFACE>
optionally assigned	CIP RECORD or USER TYPE	
is assigned to	TRANSITIONS	
is associated to	OUTPULSES	

The set of inpulses of a process represent process triggers that are activated by outpulses emitted by other processes of the same cluster (synchronous pulse cast interaction). Inpulses with an assigned CIP record or user type carry data, accessed by means of operations executed in the triggered state transition.

OUTPORT

is part of a	PROCESS	CIP Browser -> PROCESSES List -> <INTERFACE>
consists of	MESSAGES, MULTIPLICITY	
connected to	CHANNELS	

An outport of a process defines a set of messages that can be sent by the process. Every outport of a process must be connected to an outgoing channel (see *COMMUNICATION SUBNET*).

OUTPORT Editor

The OUTPORT Editor allows you to create outports with messages to which *CIP RECORD*S and *USER TYPE*S can be assigned.

With the <MULTIPLICITY> command of the OUTPORTS List menu you open an editor for specifying an outport array consisting of several outport instances. Outports arrays must be used when the process is connected to channel arrays with multiplicities different from those of the writer process.

With the <SELECTOR assignment> command you open an editor to assign *SELECTOR* functions to messages of outport arrays. SELECTOR functions are used to restrict writing of a messages to outport instances.

OUTPORT MESSAGE

is part of a	OUTPORT	OUTPORT Editor -> MESSAGES List
optionally assigned	CIP RECORD or USER TYPE, SELECTOR	
is assigned to	TRANSITIONS	

An outport message emitted in a state transition is transmitted asynchronously by the channel connected to the outport. Messages with an assigned CIP record or user type carry data, packed by means of operations executed in the emitting state transition.

OUTPORT MULTIPLICITY

is part of a	OUTPORT	OUTPORT Editor -> OUTPORTS Menu
optionally assigned	INDEX TYPES	

Defining a multiplicity of an outport creates an outport array. When connecting the outport array with a channel, the outport multiplicity must increase the multiplicity of the writer array to match precisely the multiplicity of the reader array. This match is automatically checked when a model verification is carried out.

MULTIPLICITY Editor

The editor allows you to specify an index tuple defining the multiplicity of the outport by assigning *INDEX TYPE*S. An index type can be used only once per outport.

Connection conditions are checked automatically before code generation:

Every outport index type must be contained in the channel array multiplicity, but must not be contained in the writer array multiplicity.

MESSAGE SELECTOR Assignment

is part of a	MESSAGE	OUTPORT Editor -> OUTPORTS Menu
assigned	SELECTOR	

An outport message assigned to a transition is written to all outport instances when the transition is executed. You can restrict writing of a message to the instances of an outport using *SELECTOR*S.

MESSAGE SELECTOR Assignment Editor

From a switchable list you can assign N-SELECTORS or MANY-SELECTORS of the process to the messages of an outport array.

The selection range of a selector must match the index range of the outport. This match is checked when a model verification is carried out.

OUTPULSE

is part of a	PROCESS	CIP Browser -> PROCESSES List -> <INTERFACE>
optionally assigned	CIP RECORD or USER TYPE	
is assigned to	TRANSITIONS	
is associated to	INPULSES	

Outpulses represent process events that are used to trigger other processes of the same cluster (synchronous pulse cast interaction). Outpulses with an assigned CIP record or user type carry data, packed by means of operations executed in the emitting state transition.

GATE

is part of a	PROCESS	CIP Browser -> PROCESSES Menu -> <GATES>
refers to	STATE INSPECTION and it's TRUTH TABLE	
is used in	CONDITIONS, OPERATIONS, DELAYS, INQUIRIES, SELECTORS, FUNCTIONS	
is assigned to	TRANSITIONS	

A Gate is a process interface for state inspection connections to other cluster processes. It must graphically be connected to a state inspection connector in the *INSPECTION NET*. At runtime a connected gate represents a boolean value indicating wether the inspected processes are in particular states. Gates are usually assigned as guards in a *SWITCH*.

The GATES list of the process shows the gate interfaces and gives access to the connected inspection *TRUTH TABLE*.

Gates can either be created in the GATES list of the process or by graphically connecting a state inspection connector in the inspection net.

Use of gate names in C-code constructs

Gate names are used as gate MACRO calls:

Within a process gates can be logically combined in conditions e.g:

```
            aGate1()|| aGate2()
```

giving a boolean value representing the combination of the state inspection results.

CONDITION

is part of a	PROCESS
consists of	an ANSI C integer expression
is assigned to	TRANSITIONS of SWITCHES

A condition is an ANSI C integer expression which can be assigned as guard to transitions of non-deterministic branchings in the *SWITCH* Editor.

Valid CIP MACROS

ID (instance identifier in the case of a process array)

STATUS (read access to state vector)

IN (read access to input data)

TIME (current cluster time in TICKS)

EXCEPTION (aborting the current state transition)

(see also <*CIP MACROS*>)

OPERATION

is part of a	PROCESS	CIP Browser -> PROCESSES Menu -> <OPERATIONS>
consists of	an ANSI C compound statement	
is assigned to	TRANSITIONS	

An operation is an ANSI C compound statement which can be assigned to state transitions (see OPERATION Assignment of the *MODE* Editor).

Valid CIP MACROS

SELF (read and write access to state vector)

STATUS (read access to state vector)

ID (instance identifier in the case of a process array)

IN (read access to input data)

OUTPULSE (write access to outpulse data)

OUTMSG (write access to outport message data)

TIME (current cluster time in TICKS)

EXCEPTION (aborting the current state transition)

(see also <*CIP MACROS*>)

FUNCTION

is part of a	PROCESS	CIP Browser -> PROCESSES Menu -> <FUNCTIONS>
consists of	an ANSI C function	
is used in	CONDITIONS, OPERATIONS, DELAYS, INQUIRIES, FUNCTIONS, SELECTORS	

Functions can be called in the code of all C-code constructs of the process. A function is edited as ANSI C function. The edited name of the function does not need to be the same as the component name. When you call a function you must use the edited name.

Valid CIP MACROS

ID (instance identifier in the case of a process array)

STATUS (read access to state vector)

TIME (current cluster time in TICKS)

EXCEPTION (aborting the current state transition)

(see also <*CIP MACROS*>)

INQUIRY

is part of a	PROCESS	CIP Browser -> PROCESSES Menu -> <INQUIRIES>
consists of	an ANSI C function	
is used in	CONDITIONS, OPERATIONS, DELAYS, INQUIRIES, SELECTORS	

Inquiries are process-local functions which can be called by other processes of the cluster to get information about the current state vector of the inspected process. State vector inspections must be declared graphically in the pulse cast net by corresponding state vector connections.

INQUIRY Editor

With the INQUIRY Editor you can browse through the inquiries of all processes of a cluster. The editor is opened from the CLUSTERS or PROCESSES List menu of the CIP Browser.

Inquiries are edited as ANSI C functions. The function header is displayed by the tool. The return value is always of an integer type.

The parameter list and the function body are defined in two separate text editors.

Valid CIP MACROS

The values of a single process's own state vector are referenced by way of the CIP macro STATUS (e.g. STATUS.aVariable).

In process arrays read access to the state vectors of the individual instances is indexed (e.g. STATUS[index1][index2]...[indexN].aVariable)

TIME (current cluster time in TICKS)

(see also *CIP MACROS*).

Use of inquiry names in C-code constructs

Inquiry names are used in inquiry calls:

`MyProcess.anInq()` calls the inquiry *anInq* of the process MyProcess.

PROCESS MULTIPLICITY

is part of a	PROCESS	CIP Browser -> PROCESSES Menu -> <MULTIPLICITY>
optionally assigned	INDEX TYPES	

The multiplicity of a process array is specifed by an index tuple consisting of one ore more assigned *INDEX TYPE*S. An index type can be used no more than once per process.

Connection conditions are checked automatically before code generation:

> Process array writing to singular channel or channel array:
>
> > Each index type of the channel multiplicity must be an index type of the writer array (many-to-one) or of the relevant outport array (one-to-one).
>
> Process array reading channel array:
>
> > The reader and the channel multiplicities must be specified by the same index types (one-to-one).
>
> Process array interacting with singular process or process array:
>
> > On instance level, pulse propagation among process arrays takes place as multi-cast (one-to-many).

SELECTOR

is part of a	PROCESSS	CIP Browser -> PROCESSES Menu -> <SELECTORS>
integer attribute	N (for N-SELECTORS only)	
consists of	an ANSI C function body	
is assigned to	PULSE TRANSLATIONS, OUTPORT MESSAGES	

Selectors are selection functions which allow to restrict the transmission of pulses and messages to process arrays. The pulse and message transmission among process arrays is determined at instance level by the principle of qualified linking. That is, with regard to the index types occurring at both, the sender and the receiver, transmission is one to one; with regard to the non-common index types, transmission is automatically all-to-all.

All-to-all links can be restricted by selectors:

> Pulse transmissions are restricted by assigning selectors to pulse translations.
>
> Message transmissions are restricted by assigning selectors to outport messages.

SELECTOR Editor

A selection range for a selector is defined by assigning *INDEX TYPE*S from the INDEX TYPES List in the right-hand section of the editor, or by <adjust> in the *SELECTOR* Assignment Editor. The choice made determines the parameters of the selection function. The function name is also generated by the tool.

A pulse selection range must consist precisely of those index types of the receiver that are not index types of the sender. A message selection range must match the index range of the outport array. These requirements are automatically checked at code generation time (model verification)

The switchable SELECTORS List allows you to edit two types of selector:

N-SELECTOR

> An assigned N-SELECTOR is called no more than N times by the generated system during the transmission, to select a receiver instance or an outport instance respectively (parameter values). The input parameter N indicates the number of the current call. If the selection function returns 1, it is called again by the system, unless

N calls have already occurred. Returning 0 terminates the selection procedure for the current transmission.
The constant attribute N of an N-SELECTOR is defined by the menu command <define N> in a text prompter. Default value of N is 1.

MANY-SELECTOR

An assigned MANY-SELECTOR is called by the generated system precisely once for each instance of the specified selection range during the transmission. The input parameters contain the selectable index values of the current instance. If the current instance is to participate in the transmission the SELECTOR must return the value 1; otherwise 0.

Valid CIP MACROS

STATUS (read access to the state vector of the sender process)

ID (instance identifier if the sender is a process array)

STATUS[ID.indexType1]...[ID.indexTypeN] (read access to the state vector of the sending instance of a process array)

TIME (current cluster time in TICKS)

(see also *CIP MACROS*)

PULSE SELECTOR Assignment

is part of a	PULSE TRANSLATION	CIP Browser -> PROCESSES Menu
assigned	SELECTOR	

Pulse transmissions from sender arrays can be restricted by assigning a selector function.

PULSE SELECTOR Assignment Editor

You can view the PULSE TRANSLATIONS relating to a specific RECEIVER by selecting it from the RECEIVERS List in the left-hand section of the editor.

The switchable SELECTORS List of the sender is located in the right-hand section.

Assigning a *SELECTOR* to a translation:

Pre-select a TRANSLATION and a SELECTOR.

Assign by clicking on the assignment button.

The assigned SELECTORS appear in the TRANSLATIONS List.

An assignment is deleted with the deassignment button as standard.

You specify the selection range of a SELECTOR in the SELECTOR Editor of the sender process by assignment of *INDEX TYPE*S. However, you can in the SELECTOR Assignment Editor automatically adjust the selection range of the selected SELECTOR to the index range of the selected RECEIVER by activating the <adjust> command in the SELECTORS List menu of the SELECTOR Assignment Editor.

Consistency

The selection range of a SELECTOR assigned to a TRANSLATION must consist precisely of those index types of the receiver that are not index types of the sender. This requirement is automatically checked when a model verification is carried out.

Necessary SELECTOR assignment for many to many Pulse Cast:

The effective pulse transmission between two process arrays must always be one-to-one or one-to-many (many-cast). This is safeguarded if all index types of the sender are index types the receivers.

If a sender has an index type which is not an index type of a receiver, the pulse transmissions are many-to-one or many-to-many; that is, the individual receiver instances can potentially be multiply activated.

EXTENSIONS

is part of a	PROCESS	CIP Browser -> PROCESSES Menu -> <EXTENSIONS>
boolean attributes	TIMER, CHAIN, AUTO	

Process extensions represent modeling sugar, i. e. the by the tool supported constructs could be modelled using the basic modeling elements of the CIP meta model also.

EXTENSIONS Editor (X-List of the MODE Editor)

TIMER-EXTENSION

A process is extended with a timer. A timer is set by assigning the SET TIMER symbol and a DELAY to a state transition; a timer is stopped by assigning the STOP TIMER symbol.

An expired timer enables the external invocation of a TIMEUP_ trigger.

CHAIN-EXTENSION

A process is extended with a feedback mechanism. A chain is set by assigning the SET CHAIN symbol to a state transition. A set chain enables the process to react on an externally invoked CHAIN_ trigger.

AUTO-EXTENSION

A process is extended with a virtual inport attached to a virtual channel, which delivers on an externally invoked AUTO_ trigger an AUTO_ message. An AUTO_ message is delivered only when the message is awaited by the process (state transition with assigned AUTO_ message).

DELAY

is part of a	PROCESS	CIP Browser -> PROCESSES Menu -> <DELAYS>
consists of	an ANSI C integer expression	
is assigned to	TRANSITIONS with assigned SET TIMER	

Delays are used in processes with specified TIMER EXTENSION for setting the timer delay. A delay is an ANSI C integer expression, defining a delay as number of TICKS.

The duration of a TICK is defined in the implementation of a CIP model: TICKS are delivered to a CIP unit by invocation of the standard TICK_ trigger.

Valid CIP MACROS

SELF (read and write access to state vector)

STATUS (read access to state vector)

ID (instance identifier in the case of a process array)

IN (read access to input data)

TIME (current cluster time in TICKS)

EXCEPTION (aborting the current state transition)

(see also <*CIP MACROS*>)

MODES

The modes of a process define alternative reactive behaviors of a process. Modes are specified as state transition diagrams, describing state transitions on the common process state space.

MODE

is part of a	PROCESS	CIP Browser -> MODES LIST
consists of	TRANSITIONS, SWITCHES	
is associated to	STATES of MASTER PROCESSES	

Every mode of a process is based on the set of states. inpulses, outpulses, inport and outport messages of the process. If a process has more than one mode, the active mode at run time is determined by the current state of one or more master processes (mode setting). The masters of a slave are defined in the *MODE CONTROL NET* Editor. If a process with several modes has no master, the active mode is defined by initialization.

TRANSITION

is part of a	MODE	CIP Browser -> MODES Menu -> <MODE graph>
identifier	TRANSITION NUMBER	
assigned	INPORT MESSAGE or INPULSE	
optionally assigned	OUTPULSE, OUTPORT MESSAGES, OPERATIONS, DELAY	
is node of	MODE GRAPHconnected to	STATES

State transitions represent elementary reactive process steps. Transition boxes are created graphically and connected to a pre- and a postate. The appearing transition number is an identifier given by the tool.

Transitions are triggered by inport messages or inpulses. Every transition can emit an outpulse and one message per outport. Operations assigned to a transition are executed sequentially.

If several state transitions with a common pre-state are triggered by the same input, the reaction of the state machine is not determined. In order to make the execution of the state machine deterministic, mutually exclusive conditions have to be assigned to the transitions of the branching. A transition branching with assigned conditions is termed *SWITCH*).

Transition Header Line

Special characters indicate invisible assignments:

O: OPERATIONS,   T: SET TIMER,   S: STOP TIMER.   C: SET CHAIN

W: invisible WRITES – If you assign more then one outport message to a transition, the message with the lowest port number only is displayed in the transition box.

Moving the selection cursor popups the invisible specification.

MODE GRAPH Editor

The MODE Editor allows you to construct state transition diagrams.

Tool Bar

Creating states:
place cursor; click; enter state name; place label; click.

Creating transitions:
place cursor; click.

Deleting states and transitions:
click on element to be deleted.

Interconnecting states and transitions:
press the selection button within the first element to be connected;
drag the cursor to the second element to be connected and release.
.

Disconnecting states and transitions:
press the selection button within the first element;
drag the cursor to the connected element and release.

Initialization:
place the token cursor on the intended initial state.
.

Selecting or moving:
click or drag.
multiple selection: shift-key + click or drag a selection window.
double click to open an editor for the selected element

Assignment of List Elements

Input and output symbols denoting pulses and messages of the process interface are assigned from the switchable assignment list at the right-hand section. If several inports or outports are specified, port numbers are used to qualify assigned messages. Data types can be assigned to pulses and messages in the *INTERFACE* Editor of the process which can be opened directly from the corresponding assignment list.

Input and output symbols concerning process extensions are assigned from the EXTENSIONS List activated by the X-Button.

Assigning a list element:
select transition(s);
select list element;
click assignment button.

Deassignment a list element:
select transition(s);
switch to corresponding element list; click button.
(special: EXTENSION elements must be selected correspondingly in the X-List)

Non-determinism

If the same input triggers more than one transition from the same state, the state is marked in gray (non-determinism). The *SWITCH* Editor opened from the graphic state menu allows you to assign *GATE*S or *CONDITION*S to the transition branching.

Operation Assignment

Operations defined in the *OPERATION* Editor of the process can be assigned to transitions. The OPERATION Assignment Editor of a *MODE* is opened from the graphic transition menu.

OPERATION Assignment Editor

is part of a	TRANSITION	CIP Browser -> MODES Menu -> <OPERATION assignment>
assigned	OPERATIONS	

*OPERATION*S, defined as ANSI C compound statements, are assigned to state transitions to access process variables, input and output data.

The tool supports two variants of the OPERATION Assignment Editor.

Working without execution groups:

This variant is sufficient when no output messages with data are used, i.e. all the output messages used in the state transitions of a mode have no data type.

Working with execution groups:

This variant is necessary when output messages with data types are used. Operations writing to the data fields of a particular output message are assigned into a correspondingly associated execution group. Such operations use the CIP Macro OUTMSG to access the data fields of outport messages (see CIP Macros).

You can change the variant of an opened editor with the <use execution groups / no execution groups> command of the assignment area menu. However, in the case of modes with state transitions using output messages with data types, the variant supporting execution groups is obligate. Furthermore, if operations are assigned into a *writing* or into the *after writing* execution group the variant can't be changed either.

SWITCH

is part of a	MODE	CIP Browser -> MODES Menu -> <SWITCHES>
assigned	GATES, CONDITIONS, OPERATIONS	
refers to	TRANSITIONS of a non-deterministic branching	

If several state transitions with a common pre-state can be activated by the same input, the reaction of the state machine is not determined. In order to make the execution of the state machine deterministic, mutually exclusive guards have to be assigned to the transitions of the non-deterministic branching. A guard is either a Gate or a Condition. A transition branching with assigned guards is termed SWITCH.

SWITCH Editor

The SWITCH Editor allows you to assign guards (i.e. *GATE*S or *CONDITION*S) to the transitions of non-deterministic transition branchings. Gates are specified in the *INSPECTION NET* of the cluster, conditions are defined as ANSI C expressions in the condition editor of the process. A switch is identified by a state-input pair. The SWITCH Editor is correspondingly realized as a browser on state-input pairs. If you select a state-input pair, the transition number and the poststate of the transitions of the switch are displayed in the middle list. At the right hand side of the editor the switchable GATES/ CONDITIONS List of the process is displayed. This list contains always a standard ELSE_ guard which can be assigned to one transition of a switch. The ELSE_ guard represents the boolean complement to the guards assigned to a switch.

Operations assigned to a switch are executed before the guards are evaluated. The Editor is opened with the menu command of the SWITCH List.

CIP MACROS

CIP MACROS are used in C-code constructs to access process variables and data fields of messages and pulses. In the case of process arrays, the ID macro delivers the index values of the active process instance. The TIME macro is used to access the current cluster time. The special macro EXCEPTION allows to abort an activated state transition of a process.

STATUS

Read access to the state vector of a process.

Used in CONDITIONS, FUNCTIONS, INQUIRIES and SELECTORS.

STATUS references the state vector structure of a process instance. The state vector structure stores the current MODE, the current STATE and the variable values of the process instance.

Examples

`STATUS.MODE`	reading the current mode (enumeration type value)
`STATUS.STATE`	reading the current state (enumeration type value)
`STATUS.myVariable`	reading the process variable *myVariable*
`STATUS[3][5].myVariable`	reading the process array variable *myVariable*

SELF

Write access to the state vector of an active process.

Used in OPERATIONS, DELAYS and FUNCTIONS.

SELF references the state vector structure of the active process instance.

Example

`SELF.myVariable`	reading and writing to the process variable *myVariable*

ID

Read access to the index value of an active process array instance

Used in OPERATIONS, DELAYS, FUNCTIONS, CONDITIONS and SELECTORS.

The index values of an active instance of a process array are contained in an identifier structure and are referenced by the corresponding index type name.

Example

`ID.anIndexType`	reading the index value of the array dimension defined by index type *anIndexType*

IN

Read access to the data of an input message or an inpulse.

Used in OPERATIONS, DELAYS and CONDITIONS.

IN references the data field of the received inpulse or inport message. The data type of the data field is the data type assigned to the inpulse or inport message.

Examples

`IN >= 24`	reading the data of an inpulse or inport message with an assigned user type `typedef int myInt;`
`IN.aField >= 24`	reading a data field of an inpulse or inport message with an assigned CIP record containing a field *aField*.

OUTPULSE

Write access to the data of an outpulse.

Used in OPERATIONS.

OUTPULSE references the data field of the outpulse to be emitted. The data type of the data field is the data type assigned to the outpulse.

Examples

`OUTPULSE = 24;`	writing data to an outpulse with an assigned user type `typedef int myInt;`
`OUTPULSE.aField = 24;`	writing data to an outpulse with an assigned CIP record containing a field *aField*.

OUTMSG

Write access to the data of an output message.

Used in OPERATIONS

OUTMSG references the data field of an outport message to be sent. In order to qualify which output message is concerned, operations are associated to process outports (operations assigned to specific execution groups). The data type of the data field is the data type assigned to the outport message.

Examples

`OUTMSG = 24;`	writing data to an outport message with an assigned user type `typedef int myInt;`
`OUTMSG.aField = 24;`	writing data to an outport message with an assigned CIP record containing a field *aField*.

VARIABLE

Initialization of process variables.

Used in INIT OPERATIONS.

VARIABLE is used for initializing process variables and variable fields with USER TYPE.

Example

`VARIABLE = 24;`	initializing a process variable with an assigned user type `typedef int myInt;`
`VARIABLE.aField = 24;`	initializing a process variable with an assigned CIP record containing a field *aField*.

TIME

Read access to the tick counter of a cluster.

Used in all C-code constructs.

TIME is a cluster variable which contains the current cluster time in TICKS. The time variable is of type `unsigned long`.

Example

`TIME >= 24;`	reading the current cluster time

EXCEPTION

Return statement to abort a state transition.

Used in OPERATIONS.

The EXCEPTION macro is used in transitions activated by input message. It leads to cancellation of the activated cluster transition and generates a context error.

The macro is used when the validity of an input message needs to be verified by way of its data.

Example

`if (IN.floor != SELF.nextFloor){`	aborting a state transition in case of an
`    EXCEPTION;`	inport message with an invalid floor number
`}`	

IMPLEMENTATIONS

The implementation browser (Implementations button) serves to generate C-code for the implementation of a CIP model. For this, the set of clusters is partitioned into concurrent CIP units. A collection of C-modules is created for each CIP unit.

IMPLEMENTATION

is part of a	SYSTEM	Implementation Browser -> IMPLEMENTATIONS List
consists of	CIP UNITS, IMPLEMENTATION INCLUDE	

Implementing a CIP model means connecting generated software components to a simulator of the environment, to a test bed of for a specific part of the system or to the real environment of the target system.

For the implementation a CIP model is partitioned into CIP units. The code generator of the tool transforms CIP units automatically into concurrently executable software components.

A CIP unit consists of a CIP shell defining the interface of the generated component and of a CIP machine modeling its reactive behavior:

A CIP shell is specified by a set of input and output channels.

A CIP machine is specified by a set of clusters.

CIP shell and CIP machine are specified independently. The code for a CIP unit is correspondingly produced in two separated generation steps. The code for a CIP machine can be generated only, when all external channels of the CIP machine appear also in the corresponding CIP shell. The set of external channels of a CIP machine consists of its source and sink channels, of its controlled channels and of the local channels which have been optionally declared as external. A CIP shell can be adjusted to its CIP machine automatically.

The generated code for a CIP machine behaves as reactive data object encapsulated within the CIP shell. The code of the CIP shell is a structure of function pointers corresponding to the input and output channels of the CIP unit. A CIP machine is activated by function calls through the input interface. The machine reacts back calling output channel functions implemented by the user. The CIP machine can be triggered also for pending activities such as timer activation or release of internally buffered messages.

In order to construct the connection of a CIP unit to its environment you need the generated CIP shell code only. In fact, you don't even need to specify a CIP machine to develop an executable connection.

The specification of a shell can be locked in the tool. This means that the shell definition and the messages of the shell channels can't be changed as long as the shell is locked. Locking a shell stabilizes the generated interface because the generated code is always the same.

Further information concerning the embedding of CIP units can be found in the CIP Tool manual.

IMPLEMENTATION INCLUDE

is part of a	IMPLEMENTATION	Implementation Browser -> IMPLEMENTATIONS Menu
consists of	C or C++ source code statements	

The edited text is inserted in all generated shell header files of the implementation during C-code generation.

CIP UNIT

is part of a	IMPLEMENTATION	Implementation Browser -> CIP UNITS List
consists of	CIP MACHINE, CIP SHELL, UNIT INCLUDE	
attributes	code options	

An implementation consists of one or several CIPþUNITS, each composed of a CIPþSHELL and a CIP MACHINE.

For consistent CIPþUNITS, reactive software components (C-code) can be generated. The code for the shell and the machine of a CIPþUNIT is generated individually.

UNIT INCLUDE

is part of a	CIP UNIT	Implementation Browser -> CIP UNITS Menu
consists of	C or C++ source code statements	

The edited text is inserted in the generated shell header file of this unit during C-code generation.

CIP SHELL

is part of a	CIP UNIT	Implementation Browser -> CIP SHELL Field
assigned	CHANNELS	

A CIP SHELL is specified as a sets of input and output *CHANNEL*S defining the interface of the CIP UNIT.

CIP MACHINE

is part of a	CIP UNIT	Implementation Browser -> CIP MACHINE Field
consists of	CIP MACHINE INCLUDE	
assigned	CLUSTERS	
attribute	buffer size	

A CIPþMACHINE is specified as a set of *CLUSTER*S determining the behavior of the CIPþUNIT.

The CIP SHELL for a specified CIP MACHINE can be generated automatically (use <adjust SHELL> of the CIP MACHINE Menu).

CIP MACHINE INCLUDE

is part of a	CIP MACHINE	Implementation Browser -> CIP MACHINE Menu
consists of	C or C++ source code statements	

The edited text is inserted in the generated machine header file during C-code generation.

CODE OPTIONS

is part of a	CIP UNIT	Implementation Browser -> CIP UNITS Menu
attributes	OPTIONS	

Code options allow you to configure the generated CIP UNIT code.

Code Language

The default code language is C.

Optional code generators for C++ and java are available.

Options affecting the CIP shell code (CIP Shell tab)

These options have an effect on the generated C-code of the CIP shell. They can't be changed if the CIP shell is locked.

use channel interface

The shell interfaces contain a function with a parameter name for each channel. If needed, aditional parameters are inserted for mesage data and channel instance.

use message interface

The shell interfaces contain a function without a name parameter for each message. If needed, aditional parameters are inserted for mesage data and channel instance.

use postfix

When the *use postfix* option is set, two different postfixes are possible:

unit postfix

all identifiers of C-code objects existing once per unit will have the unit name as postfix, as for instance IN_myUnit().

user postfix

user can define a string that is used as postfix to all identifiers of C-code objects existing once per unit, as for instance IN_ABC_().

If the option is not set, there will be no postfixes, as for instance IN_()

all types in shell

all used types are declared in the CIP shell header file.

enable PENDING_ information

The CIP SHELL data structure is extended by status variables informing about pending CIP MACHINE activities like expired timers or messages stored in the local communication buffer for internal channels.

call input error function

A user implemented input error function is called when an input error occurs.

Options affecting the error function interface (CIP Error tab)

These options have an effect on the generated C-code of the error function interface only. They have no effect on the CIP shell code. The error functions called due to enabled code options must be implemented by the user.

call context error function

If a message is not awaited by the receiver process, the activated CIP machine calls the context error function.

call buffer warning function

If the buffer for internal message channels becomes full, the activated CIP machine calls the buffer warning function.

call buffer error function

If an overflow of the buffer for internal message channels occurs, the activated CIP machine calls the buffer error function. The message concerned is trashed.

call pulse selection error function

If an index value of a pulse selection does not mach its index range, the activated CIP machine calls the pulse selection error function.

call message selection error function

If an index value of a message selection does not mach its index range, the activated CIP machine calls the message selection error function.

Options affecting the trace interface (Trace tab)

Traced code allows to inspect the state and the internal reaction of a CIP unit at run time.

traced code

When the traced code option is set the generated code is extended with high-level debugging functions. The running code can be triggered to output information about occurring cluster transitions. The user is required to attach a string input and a string output function at the generated trace interface.

call USER TYPE input functions

The generated cip code calls an input function

```
int fINTPUT_aUserType (aUserType *field)
```

every time input data for a *USER TYPE* is needed.

If the option is set, these functions must be provided by the user.

If the option is not set, dummy input functions for user types are generated by default but these functions leave the values undefined.

call USER TYPE output functions

the generated cip code calls the output function

```
int fOUTPUT_aUserType (aUserType *field)
```

every time output date for a *USER TYPE* must be presented.

If the option is set, these functions must be provided by the user.

If the option is not set, dummy output functions for user types are generated by default but these functions cannot display values.

name dictionary files

The dictionary access functions can be used to display the identifiers of enumeration constants. The constant names are returned as character strings.

Miscellaneous Options (Advanced tab)

enable MODE inspection

The option allows to define inquiries depending on the active mode of an inspected process. (See CIP MACRO MODE)

no static initialization

All cip machine variables are initialized at run time when the initialization function fINIT_ is called.

'unsigned char' for 'enum'

For state, input and output symbols unsigned char constants are generated instead of enumerated integer constants. The option is useful for implementations running on 8-bit processors.

’unsigned long’ for delays

The delays for the Set Timer function are defined as unsigned long. Thus the valid range is 0 to maxval(unsigned long).

’unsigned int’ for delays

The delays for the Set Timer function are defined as unsigned int. Thus the valid range is 0 to maxval(int).

’unsigned char’ for delays

The delays for the Set Timer function are defined as unsigned char. Thus the valid range is 0 to 255.

Generated source code files for a CIP unit

For a detailled description of the contents of the individual files see
CIP Tool 5 User Manual; Interface Description.

aUnitY.c	The *animation file* contains the main function as well as input and output functions. This file is generated only for animations.
sUnitY.c, sUnitY.h	In the *CIP shell file* the interface objects of the CIP unit are defined as structs of function pointers for channel or message functions. Constants, datatypes and enumerations used for the interface are declared in the shell header file.
mUnitY.c, mUnitY.h	The *CIP machine module* contains the functions existing once per CIP unit such as global input- and output functions or timer- and chain handling functions.
ProcessU.c, ...	Each *process module* implements a process as extended state machine. C-code constructs like operations and conditions are incorporated inline in these modules.
eUnitY.c, eUnitY.h	The *error Interface* consists of a struct of function pointers for user written error handling functions. The associated datatypes and enumerations are declared in the error header file.
tUnitY.c, tUnitY.h	The *trace Interface* consists of a struct of function pointers for user written string handling functions. The associated datatypes and enumerations are declared in the trace header file.
nUnitY.c, tUnitY.h	The file contains functions and tables of names retranslating enumeration type values into character strings.

ANIMATIONS

The animation browser (Animations button) serves to generate C-code for visual and textual animations of CIP units. Visual animation are executed in the tool, textual animation are executed in the programming environment. An animation represents an implementation of a CIP unit, in which keyboard, mouse and screen replace the connection to real environment.

A animation unit is a part of a CIP model consisting of a CIP shell and a CIP machine. A CIP machine is specified as a set of clusters determining the behavior of the component while the CIP shell is specified by a set of input and output channels defining the component interface. An animation unit is specified like a CIP unit in the IMPLEMENTATION Browser used to generate software components executable on the target system. The only difference is that the CIP shell of an animation unit is automatically adjusted when the CIP machine is specified.

ANIMATION UNIT

is part of a	SYSTEM	Animation Browser -> ANIMATION UNITS List
consists of	CIP MACHINE, CIP SHELL	

CIP SHELL

is part of a	ANIMATION UNIT	Animation Browser -> CIP SHELL Field
assigned	CHANNELS	

The CIP shell of an animation unit is defined automatically when the CIP machine is specified.

CIP MACHINE

is part of a	ANIMATION UNIT	Animation Browser -> CIP MACHINE Field
assigned	CLUSTERS	
attribute	buffer size	

A CIP machine is specified as a set of clusters determining the behavior of the animation unit.

Textual Animation

In a textual animation you can activate a CIP machine in steps by entering messages from the console. The animated CIP machine responds by displaying the new machine state, the process variable values and the emitted pulses and messages.

Building a textual animation

To build a textual animation first generate the code for the animation unit.

Then compile the files and link them to an animation application (.exe).

Executing a textual animation

Start the created .exe file

On the input prompt enter the implementation name of the message to trigger the CIP machine.

The reaction of the CIP machine is displayed on the screen.

Further information about textual animation can be found in the CIP Tool manual.

Visual Animation

in a visual animation you can activate a CIP machine in steps by entering messages from a graphical control panel in the tool. The animated CIP machine responds by highlighting the activated processes in the graphic pulse cast net and by highlighting the performed transitions in the opened mode graphs. Building and executing a visual animation is described below.

Building a DLL-File for a Visual Animation

Create a project in your C or C++ programming environment to build a DLL-file.

Working with Visual C++ you create a Win32 Dynamic Link-Library.

Settings: Do not use precompiled headers.

Path restrictions of the DLL-file to be built:

The file path must not contain spaces.

Executing a Visual Animation

Layout

Menu ANIMATION UNITS -> <open> opens the pulse cast nets of the CIP unit.

From the process box menus of the pulse cast nets you open with <open> the corresponding graphic transition models.

The pulse cast net and mode graph menu contains <scale up> and <scale down> for discrete zooming of the graphic models.

Starting a Visual Animation

Menu ANIMATION UNITS -> <connect>

The opened file list allows you to search and select the built DLL-file.

Successful connection opens the animation control panel containing a message browser for the input channels of the CIP unit. Connection is not possible when you have changed your model without generating code and building a new DLL-file (Prompter. invalid DLL). However, changing the layout of graphic models only is not registered as model change.

Triggering a Visual Animation

CIP units represent event driven software components which can be animated by means of messages and standard triggers for pending transitions (expired timers, internally buffered messages).

Entering a MESSAGE

Select a message in the channel browser and click the **MESSAGE** Button at the right. Data carried by a message are entered in the **Data** edit area activated by selecting the displayed fields of the corresponding CIP record.

If the channel has multiple instances, the instance must be slected by means of the index pulldown menus.

Entering TICKS

Enter a number in the TICKS area and click the **TICKS** button.

Triggering TIMEUP

When at least one timer has expired the **TIMEUP** button is activated. Clicking the TIMEUP button triggers a pending timer of the CIP machine.

Triggering CHAIN

When at least one chain is pending the **CHAIN** button is activated. Clicking the CHAIN button triggers a pending chain of the CIP machine.

Triggering READ

When at least one message has been written to an internal channel the READ button is activated. Clicking the **READ** button triggers the CIP machine to release a buffered message.

Recording a Part of an Animation in a Message Sequence File

By pressing the **Start** button in the **Input Record** area an animation record is initiated. A file prompter allows you to create a message sequence file.

The **Break** button is used to insert break points.

The **End** button serves to terminate a record.

Playing Message Sequence Files

The **Open** button in the **Play** area allows you to load a message sequence file. The **Close** button serves to terminate the current play. During during a play you can load further message sequence files (nested execution).

Use the **Step** button to trigger the animation stepwise by the messages of the currently active message sequence file.

By pressing the **Run** button the file is executed until a brake point or the file end is reached.

The **Skip** button is used to skip the next message of the message sequence file.

The **Halt** button interrupts an ongoing execution activated with the Run button.

When the **Next** button of the **Input** area is pressed, the channel browser shows the next message of the current message sequence file.

By pressing the **Insert** button of the Input area, the channel browser for user triggered animation is activated. During a play you can trigger the animation by any message selected in the corresponding message list.

Recording the Output of a Visual Animation in an Output Record File

By pressing the **Start** button in the **Output Record** area an animation output record is initiated. A file prompter allows you to create a output log file.

The **End** button serves to terminate a record.

The check box **Include Input** enables recording of the input sequence together with the produced output of the animation in the output record file.

Stopping a Visual Animation

Menu ANIMATION UNITS -> <close> disconnects the DLL-file and closes the control panel.

The animation can be started again as long as the CIP model has not been changed essentially: You are allowed only to change the form of the graphical models of the CIP specification

Documentation of CIP Models

CIP Tool allowes you to document the CIP models on a postscript printer or in a postscript file. You can customize your own reporters defining the items to be documented. Reporters are then used to generate specific documentation for CIP models.

Built in Reporters

CIP Tool has two invisible built in reporters. These reporters named ’Standard’ and ’Graphics’ can be used to create documentations of CIP models using the **<report>** menu command of CIP objects.

Editing Reporters

The menu commands <**SYSTEMS>, <CLUSTERS>, <PROCESSES>, <MODES>, <IMPLEMENTATIONS>** and **<ANIMATIONS>** open the reporter editor for the CIP object.

In the reporter editor for SYSTEMS you can create, copy, delete, export and import your own reporters. The built in reporters are not displayed.

You can also create your own copies of the built in reporters using the menu commands **<Copy of Standard Reporter>** and **<Copy of Graphics Reporter>**.

Using Reporters

Using the **<report>** menu command or a CIP object you open a prompter in wich you can select a reporter and print or generate a documentation of the CIP object.

For more information about configuration and use of reporters please refer to the chapter **Documentation** in the **CIP Tool User Manua**l.