

# 6 *The CIP*

## *Modelling Framework*

---

A CIP model is an executable specification of a distributed reactive machine. CIP models are built by creating, composing, and connecting extended finite state machines. In this chapter we consider CIP models that consist of singular state machine instances only. Multiple state machine instances, specified as state machine arrays, are treated in Chapter X.

The compositional modelling approach is supported by CIP Tool which allows constructing CIP models by means of a system of graphic and text editors. The modelling process is supported by automated analysis and verification functions of the tool. The implementation of CIP models is automatised by code generators that produce executable software components with separately generated interfaces. Model documentation is also generated automatically in the form of graphical model reports.

The description of the various CIP modelling examples presented in this book correspond to automatically generated *model reports*. A generated model report is formal model description that is as rigorous as a model description by means of a formal specification language. For presentation and space economical purposes, the layout of the generated reports has been edited by hand.

The presentation of the modelling framework starts in section 6.1 with an overview of the general CIP model structure. Section 6.2 describes the general elements of an extended finite state machine, and how these elements can be related in the graphical behavioural specification. The subsequent sections describe the interaction mechanisms supported by the modelling framework. Asynchronous message passing between state machines of a CIP system is treated in section 6.3. Sections 6.4 to 6.6 explain cluster local interaction types that are not visible outside a cluster. The model examples presented in these sections therefore consist of a single cluster only. In addition, we recommend to reconsider the introductory MSS model variants presented in chapter 5.

### 6.1 CIP Model Structure

A CIP model consists of several concurrent CLUSTERS, viewed in Figure 32 as multiple frame of the SYSTEM frame. Each cluster is a synchronous composition of a number of extended finite state machines termed PROCESSES.

Within a cluster, three different interaction nets define configuration of interaction connections between processes. First, the PULSE CAST NET is used to specify stimuli transmission by means of multicast interaction. Second, the INSPECTION NET allows declaring state dependencies between processes. Third, the MODE CONTROL NET defines which processes determine the active mode of a process.

Each of these interaction nets defines an interaction configuration by means of interaction connectors that are linked to corresponding process interfaces. For each connection the supported interaction mechanism is specified by a mapping between the connected process interfaces.

All processes of a system can communicate asynchronously via buffered CHANNELS. Channels are created as autonomous connector objects. Source and sink channels are used to connect processes to the system environment. Communication connections are constructed graphically in the COMMUNICATION NET by linking processes and channels.

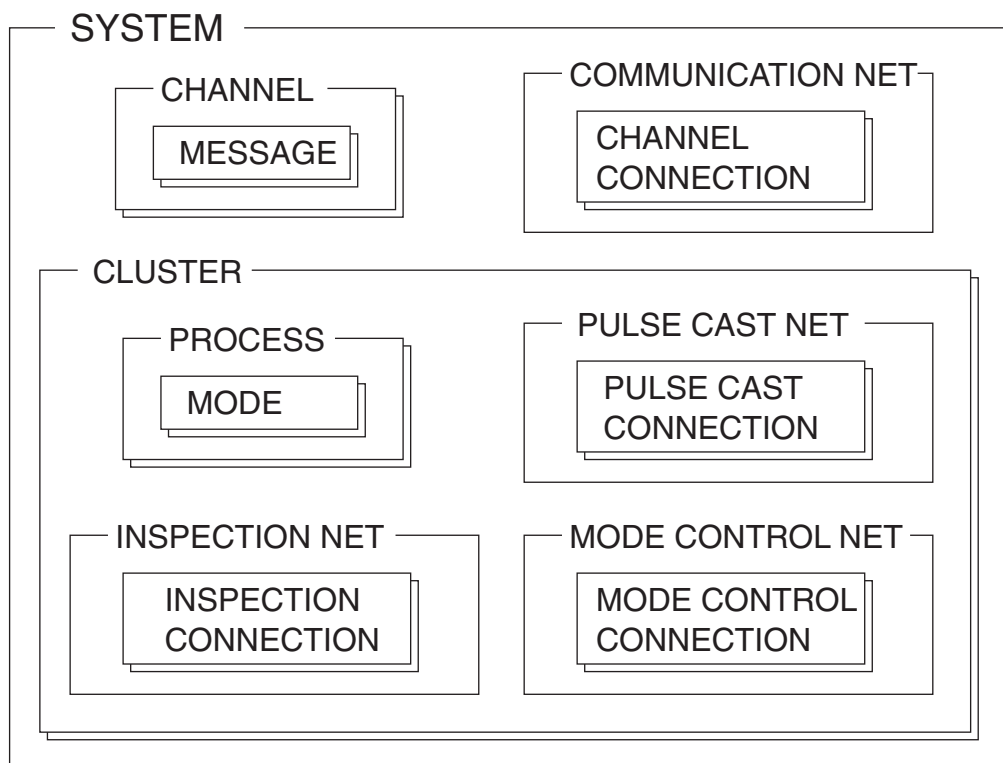


Figure 32: Components and connections of a CIP model

## 6.2 Processes – Extended Finite State Machines

A process is an event-driven extended finite state machine; that is, a plain state machine can be extended with local variables, data types assigned to input and output stimuli, conditions, and operations executed in state transitions.

### 6.2.1 Modelling Elements of a Process

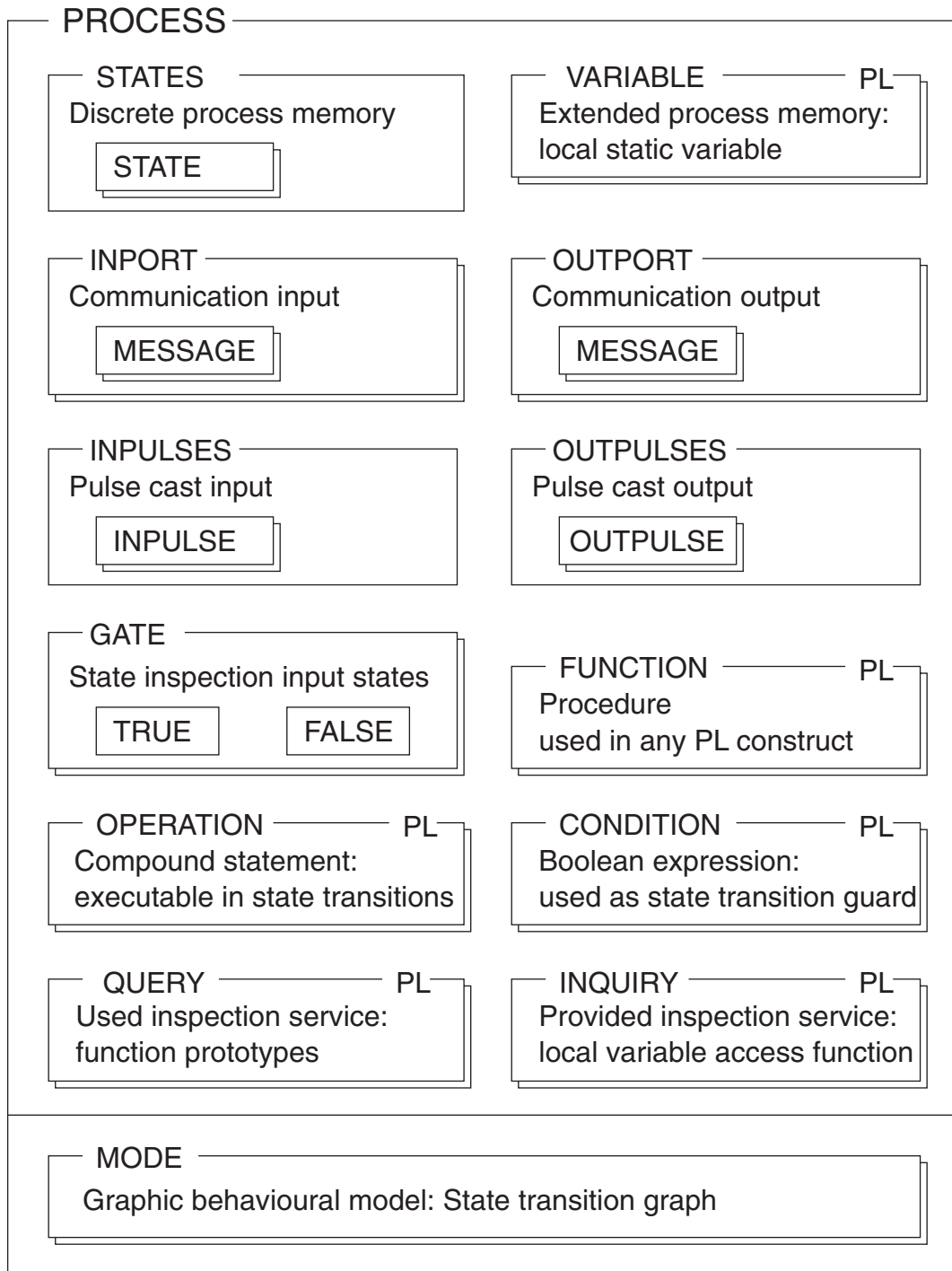


Figure 33: Process elements

Processes are true behavioural components; that is, the behaviour specification of a process is based on local process elements only, such as input and output stimuli, states and process variables. As these basic process elements are not visible by other processes, they must be related explicitly when interaction connections are constructed. The fundamental advantage of the component-based modelling approach is that a process can be used in different modelling contexts, and that local changes have no side effects in other process specifications. If we remove, for instance, a process from a consistent CIP model, the remaining processes still define a consistent executable reactive machine.

In Figure 33 above, the large upper compartment of the depicted process frame contains the basic modelling elements of a process. These are used to define the process memory, the process interfaces, and the computational extension expressed in a programming language (PL marker). These independent modelling elements are related by the process behavioural model which is described by a set of alternative state transition diagrams termed MODES.

### **Process Memory – STATES, VARIABLES**

A finite set of state symbols defines the possible values of the discrete state memory of the plain state machine. The state memory can be extended by means of a list of process variables. Process variables are declared in a programming language (PL). We use the term *state vector* to denote the current value of the process memory, i.e. the current state and process variable values.

### **Channel Interfaces – MESSAGE INPORT, MESSAGE OUTPORT**

One part of the stimuli interface of a process consists of a set of inports and a set of outports. Each of these message ports represents an interaction point for message passing interaction and must be linked to a channel as a receiving or a sending port respectively. A message port is specified by a set of message symbols designating the stimuli to be received or sent. The state machine extension is supported by data types assigned to message symbols.

### **Pulse Cast Interfaces – OUTPUTS, INPUTS**

The other part of the stimuli interface consists of an input and an output interface for pulse cast interaction with other processes of the same cluster. A sent output pulse appears as received input pulse in the processes that are connected by a pulse cast connector. The state machine extension is supported by data types assigned to pulse symbols.

### **State Inspection Interfaces – GATE, STATES**

State inspection interaction allows specification of state dependencies between the processes of the same cluster. A state inspection connection represents

boolean function that depends on the current states of the inspected processes, and that updates a boolean gate of the inspecting process. The function is defined by a truth table that specifies the boolean gate value for each state combination of the inspected processes. The gates of a process can be associated to branching state transitions to determine the execution of the branching.

### **Mode Control Interfaces – MODES, STATES**

The enabled mode of a process is determined by the current states of one or more cluster processes that are connected by a mode control connector. The interaction of a mode control connection is specified by a mode setting table which defines for each state combination of the controlling processes the enabled mode.

### **Computational Primitives – FUNCTIONS, OPERATIONS, CONDITIONS**

Computational primitives are programming language constructs that are used to extend the computational power of the plain state machine.

*Functions* are local procedures that can be called within the code of any computational primitive of the process.

*Operations* are compound statements that can be executed in state transitions. Each operation can access input data, read and update process variables and write output data.

*Conditions* are boolean expressions that are used, like gates, to determine the execution of branching state transitions.

### **Computational State Vector Inspection – QUERIES, INQUIRIES**

A query-inquiry link is a computational extensions of the state inspection interaction. The queries of a process represent a procedure call interface which is to be connected to inquiries provided by other cluster processes. An inquiry is defined as a procedure (PL) that delivers information depending on the current state vector of the inspected process.

### **Process Behaviour – MODES**

The process behaviour is defined by one or more alternative modes that are defined by graphical state transition diagrams. State transitions are triggered by input messages or pulses. Every state transition can emit one outpulse and one message per output. If in the same state several alternative state transitions are possible for the same input stimulus, the indeterminate reaction can be made determinate by assigning gates or conditions to the individual state transitions. The active mode of a process is determined by the current states of those cluster processes that interact by means of mode control interaction.

### 6.2.2 *The Plain State Machine*

As a first process example we present a plain state machine; that is, a state machine without computational extension. A plain state machine is an executable behavioural model that abstracts completely from any possible implementation on a computer. The process elements that represent computational extensions, defined as programming language constructs, are labelled in Figure 33 with a PL mark. If we do not use such process elements, the specified process is a plain state machine.

All types of interaction connections supported by the modelling framework represent interaction models that define interaction mechanisms between plain state machines. When we build a CIP model consisting of plain state machines only, we thus construct a distributed reactive machine that abstracts from any possible implementation on a computer too.

The behaviour of a process is modelled by a graphic state transition structure. Each state transition is triggered either by an input message or an inpulse. In each state transition one outpulse and one message per outport can be emitted.

An occurring input message must be expected in the current state, that is, there must exist a state transition that is triggered in the current state by this message. Otherwise, the input message is invalid and its occurrence is treated as *context error*. In the generated code the invalid message can be processed in an invoked context exception handler. An occurring inpulse, on the other hand, is valid in any state. If an inpulse is not expected it is ignored.

#### *Example: Simple Vessel Control*

The *SimpleVesselCtrl* process is used in a cluster which controls the equipment of a chemical production process. The *SimpleVesselCtrl* process controls a vessel that delivers on request a vessel filling of a certain liquid to the chemical reaction. The vessel is equipped with a bottom and a top level sensor. Two valves can be controlled to fill and empty the vessel.

The process specification of Figure 34 below contains the *LevelEvents* inport and the *ValveActions* outport which allow the process to be connected to the system environment. The input messages *Filled* and *Emptied* notify that the corresponding vessel event has occurred. The output messages *Fill*, *Drain* and *Close* are used to cause corresponding vessel actions. In addition, the stimuli interface for the pulse cast interaction with other cluster processes consists of the two inpulses *start* and *end*, and of the singular outpulse *done*. Furthermore, the *grantDraining* gate represents a boolean input interface. The gate value depends on the current states of other cluster processes (state inspection).

**PROCESS SimpleVesselCtrl**

**STATES**

draining, filling, full. empty

**INPORT LevelEvents**

MESSAGES Emptied, Filled

**INPULSES**

go, start

**OUTPULSES**

done

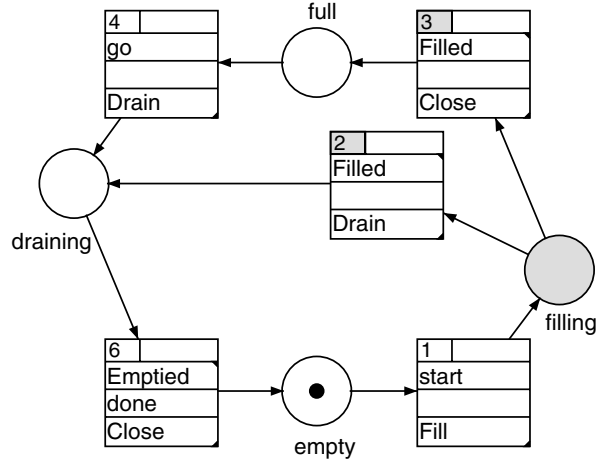
**OUTPORT ValveActions**

MESSAGES Close, Fill, Drain

**GATES**

grantDraining

**MODE normal**



**SWITCH**

STATE filling MESSAGE Filled  
 TRANSITION 2 / grantDraining  
 TRANSITION 3 / ELSE\_

*Legend:*

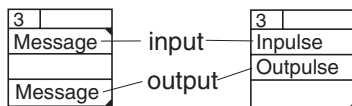


Figure 34: CIP specification of the SimpleVesselCtrl process

The *normal* mode (state transition diagram) defines the behaviour of the process. In the *empty* initial state, marked by a token, a first state transition can be triggered by a *start* pulse from an other cluster process. This transition to the *filling* state emits the *Fill* message to cause the inflow valve to be opened.

When the expected *Filled* message occurs the process reaction is indeterminate (filling state marked in grey) because two state transitions are possible. Transition 2 emits the *Drain* message to empty the vessel while the transition 3 emits the *Close* message to cause closing the vessel valves. The execution of this transition branching is determined by the *grantDraining* gate. The state transition to be enabled by this gate is specified by the SWITCH construct below the graphical model. If the *grantDraining* gate value is true, the transition 2 is executed; else the transition 3 is executed. The assigned ELSE\_ condition is a standard complementary condition that is used to complete the assignment of gates (or conditions) to the transitions of a branching.

If the vessel process branches to the *full* state, it remains in that state until another process sends the *go* pulse; the state transition triggered by the *go* impulse causes then the vessel to be emptied too.

Finally, the occurring *Emptied* message triggers the state transition back to the initial state. The emitted *Close* message causes closing the vessel valves, and the cast *done* pulse informs other processes about the delivery of a vessel filling.

### 6.2.3 The Extended State Machine

In order to support data processing and algorithmic concerns, CIP processes are specified in general as extended state machines. By means of the underlying plain state machine and its computational extension, reactive functionality can be specified on two different levels of abstraction.

Process variables values and output data are computed by means of operations when a state transition of the underlying plain state machine is executed. If the plain state machine has branching state transitions without assigned gates, the state machine can be rendered deterministic by assigning conditions to these state transitions. Conditions are formulated as boolean expressions that can depend on the current input data, on the local process variables, and on the process variables of other cluster processes if inquiry calls are used.

#### *Example: Extended Vessel Control*

The functionality of the simple vessel control example is enhanced in the following way. First, each liquid portion to be delivered consists of a requested number of vessel fillings. Second, the opening of the outflow valve can be controlled to support different flow intensity. The extended functionality has led to an elaboration of the *SimpleVesselCtrl* process. Computational extensions are defined in the C programming language. The resulting *ExtendedVesselCtrl* process is presented in Figure 35. The following process elements have been added:

**User Types.** User types are globally defined data types; the same collection of data types can be used in all processes of a CIP system. The specified *tInteger* type is used to extend the *start* impulse by an integer value which carries the ordered number of vessel fillings. The same data type is declared for the *order* and *counter* process variables. The *tFloat* data type is used to extend the *Drain* output message by a parameter value for the outflow valve.

**Inquiries.** The *fillings* inquiry can be called by a other processes (query calls).

**Queries.** The *drainFlow* query is used to update the output data field of the *Drain* message. The query is defined as C function pointer. The called inquiry and its provider is not visible in the process specification.

**Operations.** Input and output data fields as well as process variables are read and written by means of operations assigned to state transition. Operations are defined as C compound statements. **Variables.** The *order* variable is used to store the ordered number of vessel fillings, and the *counter* variable retains how much times the vessel has been filled.

**Conditions.** The *enough* condition is a C expression used to decide if the emptied vessel has delivered the requested number of vessel fillings.

USER TYPE tInteger

```
typedef int tInteger;
```

USER TYPE tFloat

```
typedef float tFloat;
```

**PROCESS ExtendedVesselCtrl**

INQUIRY fillings

```
tInteger fillings (void){
    return STATUS.counter;
}
```

STATES

draining, filling, full, empty

VARIABLES

order: tInteger, counter: tInteger

INPORT LevelEvents

MESSAGES Emptied, Filled

IMPULSES

go, start: tInteger

OUTPULSES

done

OUTPORT ValveActions

MESSAGES Close, Fill, Drain: tFloat

GATES

grantDraining

QUERY drainFlow

```
tFloat (*drainFlow) (void);
```

OPERATION storeOrder

```
{SELF.order = IN;}
```

OPERATION initCounter

```
{SELF.counter = 0;}
```

OPERATION incrementCounter

```
{SELF.counter++;}
```

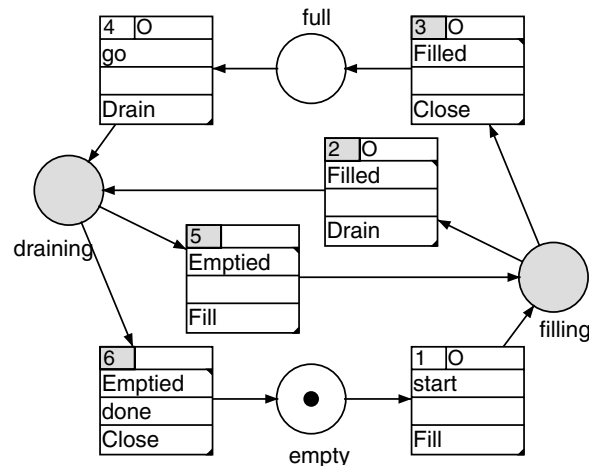
OUTPORT OPERATION giveDrainFlow

```
{OUTMSG = QUERY.drainFlow();}
```

CONDITION enough

```
STATUS.counter >= STATUS.order
```

MODE normal



SWITCHES

STATE filling MESSAGE Filled

TRANSITION 2 / grantDraining

TRANSITION 3 / ELSE\_

STATE draining MESSAGE Emptied

TRANSITION 6 / enough

TRANSITION 5 / ELSE\_

OPERATION ASSIGNMENT

TRANSITION 1 / OPERATIONS storeOrder, initCounter

TRANSITION 2 / OPERATIONS incrementCounter

OUTPORT ValveActions OPERATIONS giveDrainFlow

TRANSITION 3 / OPERATIONS incrementCounter

TRANSITION 4 / OUTPORT ValveActions OPERATIONS giveDrainFlow

*Figure 35: CIP specification of the ExtendedVesselCtrl process*

*Condition Assignment.* The *normal* mode of the *ExtendedVesselCtrl* process differs from the *SimpleVesselCtrl* process by a second state transition that is triggered in the *draining* state by the *Emptied* message. The new transition to the *filling* state causes the vessel to be refilled. This transition must be executed when the number of requested vessel fillings has not yet been delivered; otherwise the transition to *empty* state must be activated.

The two state transitions triggered by the *Emptied* message specify a further transition branching of the *ExtendedVesselCtrl* process. Conditions are used, similarly as gates, to determine the reaction on the *Emptied* message in the *draining* state. The CONDITION ASSIGNMENT list shows that the transition 6 to the *empty* state is enabled if the *enough* condition is true, else the transition 5 to the *filling* state is executed. The **STATUS** identifier used in the *enough* condition code is for reading the state vector; the general use of this standard identifier is explained below (see *State Vector Access in Operations, Conditions and Inquiries*).

If the conditions used to resolve an indeterminate reaction are based on common computations, these computations can be specified as PRE OPERATIONS. Pre operations are process operations that are assigned to a transition branching; they are executed before the assigned conditions are evaluated.

*Operation Assignment.* Operations assigned to a state transition are executed when the transition fires. We need the *storeOrder* operation to copy the requested number of vessel fillings, carried by the *start* impulse, to the *order* process variable. The *initCounter* and the *incrementCounter* operations are used to maintain the *counter* variable. The *giveDrainFlow* operation is an OUTPORT OPERATION that is used to update the *Drain* message of the *ValveActions* outport. This operation uses the *drainFlow* query to get the current parameter value from an other cluster process. The meaning of the various standard identifier used in operation are explained below also (see *State Vector and Data Field Access ...*).

State transition with assigned operations are marked in the small top info line of the transition box with an O symbol. The operations assigned to the various state transition are reported in the OPERATION ASSIGNMENT lists. If several operations are assigned to one transition, the execution order is defined by the list order.

When an outport operation is assigned to a state transition, one must in addition specify to which outport the accessed message belongs (see also **OUTMSG** identifier below). In the operation assignment list for transition 2 and 4, the assigned *giveDranFlow* operation is thus qualified by the *ValveActions* outport.

*State Vector Access in Operations, Conditions and Inquiries (C code extension)*

In order to encapsulate process variables, the state vector of a process is implemented (CIP Tool code generation) by a **struct** that contains the discrete state and the set of process variables.

**SELF**. In the C code of *operations*, the standard **SELF** identifier is used to read and write to the state vector of a process. For example, the process variable **counter** is accessed by **SELF.counter**.

*Remark:* In object-oriented state machine extensions, e. g. Java, process variables can be accessed directly by the variable name because they are implemented as instance variables of generated java objects (see CIP Tool manual).

**STATUS**. In the C code of *conditions* and *inquiries*, the standard **STATUS** identifier is used to read the state vector of a process. The **STATUS** identifier does not allow writing to process variables. The *enough* condition, for instance, compares with **STATUS.counter >= STATUS.order** the *counter* and *order* process variables.

Within an inquiry it may be necessary to formulate conditions that depend on the current process state. **STATE** is the name of the discrete process state variable. For instance, the expression **STATUS.STATE == empty** evaluates to true when the process is in the *empty* state.

*Data Field Access in Operations and Conditions (C code extension).* The identifiers described below are used to access the data fields of the input stimulus and the output stimuli of an activated state transition. The data type of the referenced data field is the data type of the corresponding stimulus (supported by the CIP Tool code generation).

**IN**. Within operations and conditions, the current input data field is accessed with the standard **IN** identifier. If the input data type is, for instance, a struct containing a component named **anInputVar**, the input variable is accessed by **IN.anInputVar**.

**OUTPULSE**. Within operations the data field of the current outpulse is accessed with the standard **OUTPULSE** identifier.

**OUTMSG**. Within outport operations the data field of the current output messages is accessed with the standard **OUTMSG** identifier. Which one of the current output message is accessed is specified when the operation is assigned to a state transition (see *Operation Assignment* above).

If no data type is declared for a particular input or output stimulus, the corresponding data field identifier is not defined.

*Inquiry Access via the Query Interface.* Inquiries of a process are local C functions (object methods in Java extensions) that deliver information which depends on the current state vector of the inquired process. Inquiries are called via the local query interface of the inquiring process. A query can be invoked in any operation, condition or inquiry of a process. Inquiries are linked to queries when state inspection interaction is specified.

**QUERY.** The **QUERY** standard identifier is used to access the local query interface of a process. A query named *myQuery* is invoked by the function call **QUERY.myQuery()**. A query, and correspondingly an inquiry, may have any number of function parameters.

*Initialisation of Process Variables.* Process variables are initialised by default to zero. Explicit process variable initialisation is supported by CIP Tool by specific INIT OPERATIONS that can be assigned to process variables.

*Process Includes.* In order to support the use of externally defined C code constructs in user type definitions, operations, conditions and inquiries, every process can be extended by an INCLUDE item. The include item is a text, that is included at the beginning of the generated process code.

*Process Arrays.* Processes can be created as multiple instances of a multi-indexed process arrays. Process arrays and the construction of interaction between multiple processes is treated in chapter XX.

*Modelling Sugar.* CIP Tool supports some standard modelling constructs such as timer setting and self triggering mechanisms. These modelling constructs are termed *modelling sugar* because they simplify the specification of some often encountered behavioural patterns.

A modelling sugar does not really extend the expressive power of the CIP modelling frame work because such a construct can always be replaced by basic modelling constructs. A timer, for instance, can be modelled by a *Timer* process that is driven by a *Tick* events. Another process can use the *Timer* process by sending a *setTimer* pulse to it. The *Timer* process in turn triggers the timer user when the specified delay has elapsed by means of a sent *timeUp* pulse.

Modelling sugars are described in section 6.7

## 6.3 Message Passing Interaction – Communication

Message passing interaction is used to transmit stimuli between processes of different clusters and between processes and the system environment. Messages are transmitted by means of asynchronous channels that interconnect sender and receiver processes.

### 6.3.1 The Elements of a Message Passing Connection

Message passing connectors are termed CHANNELS. Channels represent autonomous modelling objects of a CIP system, as indicated by the system frame of Figure 32 of section 6.1. Each channel is defined, similarly as a message port of a process, by a finite set of messages with optionally assigned data types.

Channels are linked in a COMMUNICATION NET diagram to message ports of sender and receiver processes. Figure 36 shows an example of a communication net. A channel that is not linked to any sender process is a *source channel*, one with no receiver links is a *sink channel*. Source and sink channels represent connections to the system environment.

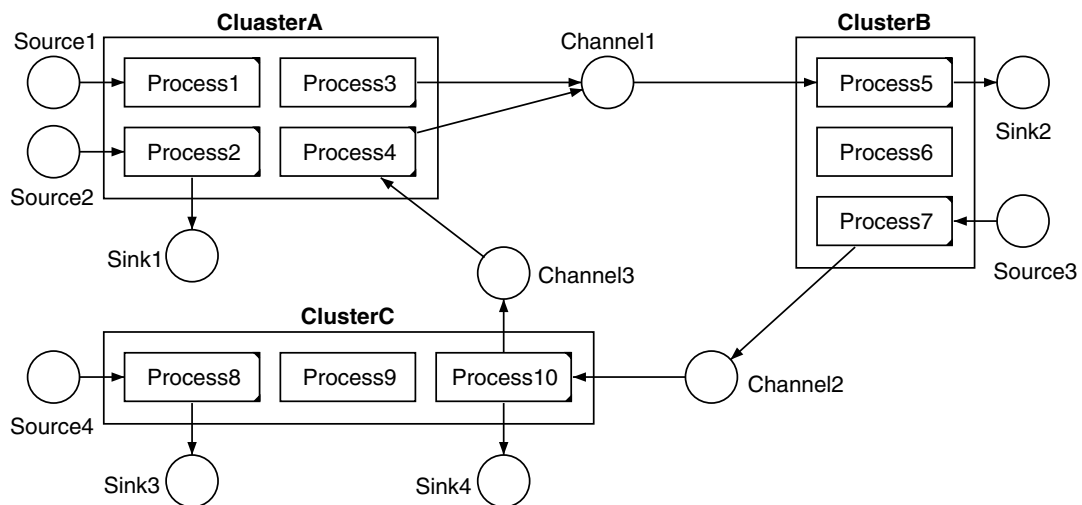


Figure 36: COMMUNICATION NET – Processes linked with channels

When a channel is linked with an outport or an inport of a process, the channel messages must be mapped to the messages of the linked process ports. Outport-channel mappings and channel-inport mappings are termed MESSAGE TRANSLATIONS.

## 6 The CIP Modelling Framework

The frame of Figure 37 shows all elements of a message passing connection. The LINKED INTERFACES are message ports that are selected when the processes and channels are connected graphically in the communication net of a CIP model. A channel can be linked with one or more outputs of different senders, but there is one link possible only with an inport of a receiver. Vice versa, each message port can participate in *one* channel link only. Source channels have no receiver links, sink channels have no sender links.

The set of MESSAGE TRANSLATIONS represents the interaction relation. As a channel must deliver each sent output messages as a unique message to the receiver inport, output, channel and inport message must be related one-to-one. Thus the number of output messages, the number of channel messages and the number of inport messages of a connection must be equal.

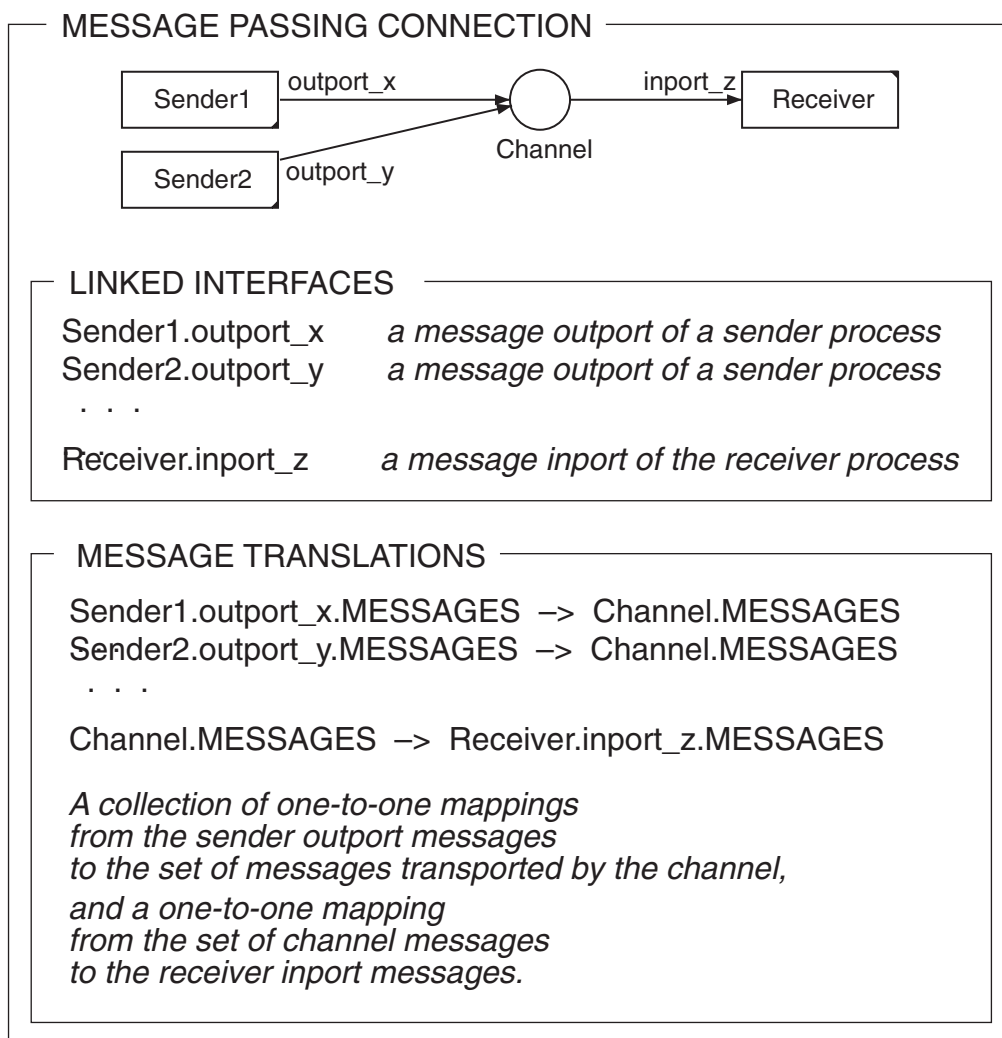


Figure 37: Elements of a message passing connection

**Remark.** Because port and channel messages correspond one-to-one, associated messages of a connection can be given a common name. In order to sup-

port ease of model readability, we shall use in the presented modelling examples equal message names for associated port and channel messages; for simplicity we shall omit explicit message translation tables. (Automatic message name adaptation for channel connections is optionally supported by CIP Tool.)

### *Extended Message Passing Interaction.*

Plain message passing interaction specifies asynchronous transmission of stimuli between processes that represent plain state machines. In order to support message passing between extended state machines, channel messages can be extended, similarly as port messages, by data types of a programming language. Data type compatibility between sent and received messages is to be ensured when output, channel and input messages are related: messages associated by message translations must have the same data type.

### *6.3.2 The Message Passing Interaction Mechanism*

A channel is an interaction connector that accepts output messages from one or more processes. The received messages are stored in a sequential fifo buffer. Messages are delivered to the linked inport of the receiver process when the corresponding cluster is not active. Channels can be used also to transmit messages between processes of the same cluster, or even to the sender itself. Source and sink channels transmit messages from and to the system environment.

Channels are active connector objects while clusters represent passive behavioural entities that are triggered concurrently. As each cluster is composed of a set of processes, an inactive cluster can be activated by an inport message of a process only. The inport messages of the cluster processes thus represent the external input interface of the cluster machine. Similarly, the output messages of the processes appear as external outputs of a cluster.

A process activated by an inport message performs a state transition, and can cause by means of pulse cast interaction (see section 6.4) a chained activation of further cluster processes. The caused chain reaction always runs to completion before the cluster can be activated again by a further inport message (run-to-completion semantics). Thus, as long as a cluster remains active due to internal interactions, the channels linked to inports are disabled from delivering messages.

If during system execution, several channels have to deliver a message to the same cluster, there is no activation order defined by the CIP model. The activa-

tion order at run-time is determined by the implementation of the involved channels. Defining the scheduling of implemented channels is part of the solution task of the connection problem, which has been separated from the functional problem of constructing reactive machines (see Chapter 3).

*About the Implementation of CIP Models.* The purpose of this preliminary note on the implementation of CIP models is to elucidate the crucial role of channels when the generated code of a CIP a model is to be linked to device interface modules. More about implementation of CIP models can be find in the dedicated chapter X.

For the implementation of a CIP model, the model is partitioned into cluster groups, and for each cluster group the tool generates an executable software component termed *CIP unit*. During system implementation the role of the channels of a CIP model is twofold. First, the modelled channels specify the connection that is to be implemented between the generated CIP units and the the system environment, as well as the interconnection of the generated CIP units. Second, the tool generates from the channels of a CIP model the complete interface of each CIP unit.

The modelled connection to the system environment (source and sink channels) is to be implemented by interface devices attached to the external processes, and by active device interface modules which connect interface devices and CIP shell. These modules trigger the CIP unit for detected events by corresponding source channel messages, and they produce the actions commanded by sink channel messages. Furthermore, channels that interconnect processes of different CIP units have to be implemented – depending on the distribution of CIP units – by means of inter task or inter processor communication artefacts. For channels that connect processes of the same CIP unit, the implementation by means of software buffers can optionally be generated by CIP Tool.

### 6.3.3 A Classical Resource Sharing Problem: *PipelineAccessControl*

The *PipelineAccessControl* modelling example addresses the classical problem of controlling the exclusive access of a single resource by concurrent clients.

#### *The PipelineAccessControl Problem – PAC*

Two plants of a distributed production process consume oil from a common pipeline branch. Only one plant at a time is allowed to consume oil from the pipeline. Each plant controls its oil consumption by opening and closing a corresponding oil valve. The oil consumption of each plant is started and stopped

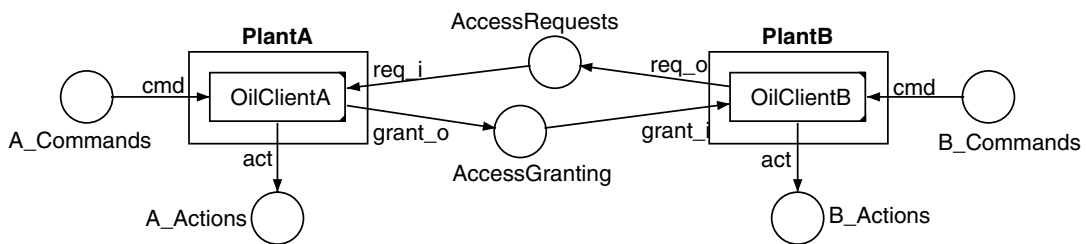
### 6.3 Message Passing Interaction – Communication

by corresponding operator commands. We suppose for simplicity that a started consumption is stopped only when oil has effectively been delivered. Furthermore, the control software of the two plants is to be implemented on two different computers.

#### A. Asymmetric Solution Based on Two Concurrent Resource Access Controllers

**SYSTEM** PipelineAccessControl\_AsymmetricSolution

COMMUNICATION NET



**CLUSTER** PlantA

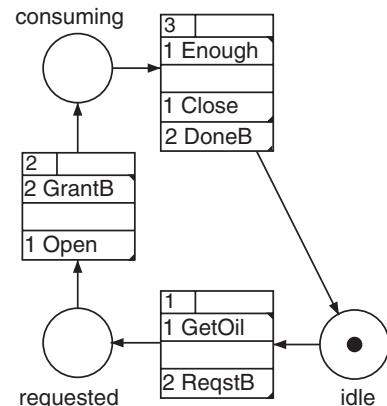
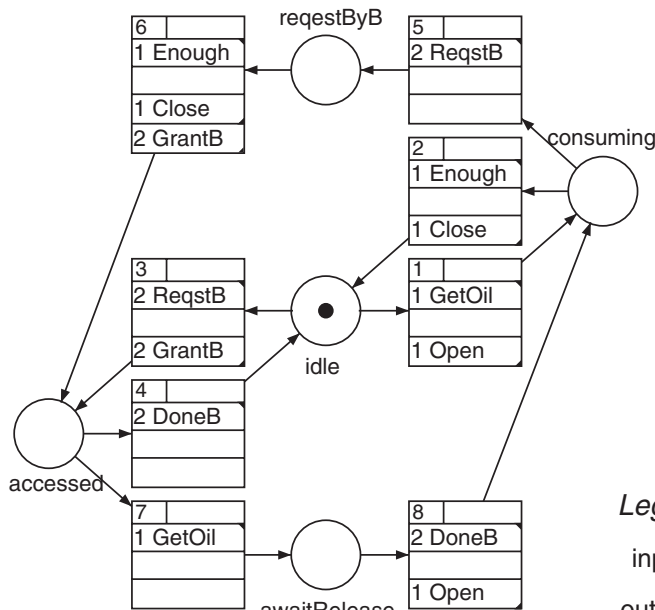
**PROCESS** OilClientA

INPORT 1 cmd  
 MESSAGES Enough, GetOil  
 INPORT 2 req\_i  
 MESSAGES DoneB, ReqstB  
 OUTPORT 1 act  
 MESSAGES Close, Open  
 OUTPORT 2 grant\_o  
 MESSAGES GrantB  
 MODE normal

**CLUSTER** PlantB

**PROCESS** OilClientB

INPORT 1 cmd  
 MESSAGES Enough, GetOil  
 INPORT 2 grant\_i  
 MESSAGES GrantB  
 OUTPORT 1 act  
 MESSAGES Close, Open  
 OUTPORT 2 req\_o  
 MESSAGES DoneB, ReqstB  
 MODE normal



**Legend:**

inport number 

3	
1	GetOil

 input message  
 outport number 

2	ReqstB
---	--------

 output message

## 6 The CIP Modelling Framework

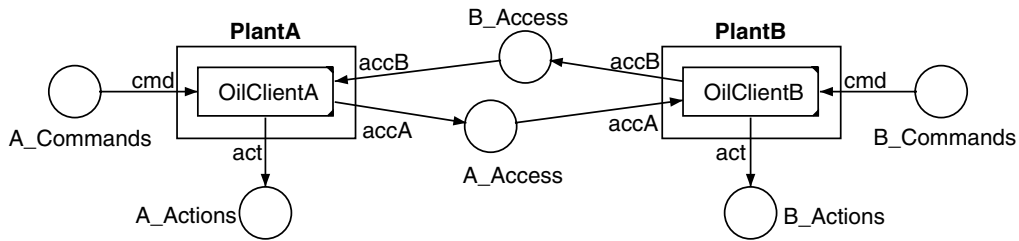
The *OilClientA* process masters the exclusive pipeline access. When the *OilClientB* process is commanded to deliver oil (*GetOil*), the process request pipeline access permission by sending the *ReqstB* message to the *OilClientA* process. When the requested permission is confirmed by the *GrantB* message, the oil valve is opened. Termination of oil consumption is acknowledged to the *OilClientA* process by sent the *DoneB* message.

*Remark.* Port numbers are used when a process owns more than one inport or outport to qualify message symbols assigned to state transitions.

### B. An incorrect solution that may run correctly

#### SYSTEM PipelineAccessControl\_IncorrectSolution

##### COMMUNICATION NET

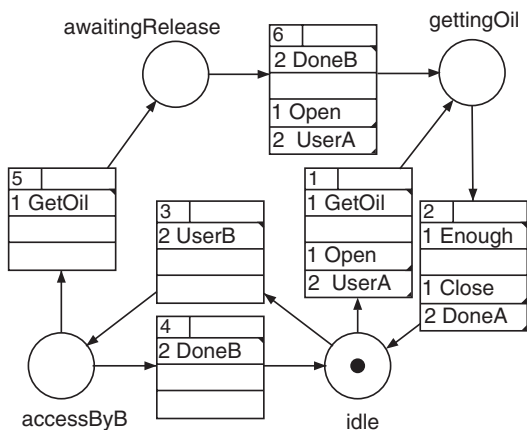


#### CLUSTER PlantA

##### PROCESS OilClientA

INPORT 1 cmd  
 MESSAGES Enough, GetOil  
 INPORT 2 accB  
 MESSAGES DoneB, ReqstB  
 OUTPORT 1 act  
 MESSAGES DoneB, UserB  
 OUTPORT 2 accA  
 MESSAGES DoneA, UserA

MODE normal

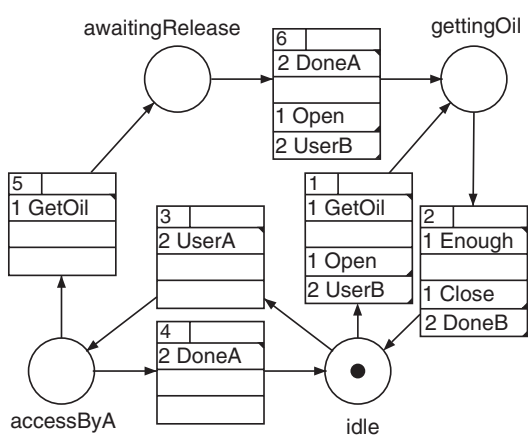


#### CLUSTER PlantB

##### PROCESS OilClientB

INPORT 1 cmd  
 MESSAGES Enough, GetOil  
 INPORT 2 accA  
 MESSAGES DoneA, UserA  
 OUTPORT 1 act  
 MESSAGES Close, Open  
 OUTPORT 2 accB  
 MESSAGES DoneB, UserB

MODE normal



The idea of the symmetric solution is that each client informs the other one when it accesses the pipeline by the *UserA*, *DoneA* and the *UserB*, *DoneB* messages respectively. However, the solution is not correct because it allows opening both plant valves at the same time. If for instance the *GetOil* commands for the *OilClientA* and the *OilClientB* process are performed simultaneously, both processes cause with their *Open* message opening their valve to start oil consumption. Although the two processes also emit the *UserA* and *UserB* message to inform the other process from accessing the pipeline, these messages have no effect because the corresponding receiver has already accessed the pipeline.

One could argue that the occurring race condition could be eliminated by implementing both clusters on a single processor, and by restricting the system behaviour by appropriate channel priorities. For instance, if internal channels *A\_Access* and *B\_Access* have a higher priority than the commands channels, the mentioned race condition effectively disappears and the solution runs correctly.

However, we refute such an approach, because the functional problem (exclusive resource access) is partly solved only by the CIP model. The correct system behaviour is enforced by subsequently restricting the channel implementation. Completing a functional model by subsequent implementation restrictions violates our basic development principle of strictly separating functional and connection concerns.

We shall discuss the incorrect solution again in the next section when pulse cast interaction is introduced. The cluster local interaction mechanism allows, due to its run-to-completion semantics, the definition of a correct functional solution that is independent of the channel implementation.

### *C. The Generic Monitor Solution*

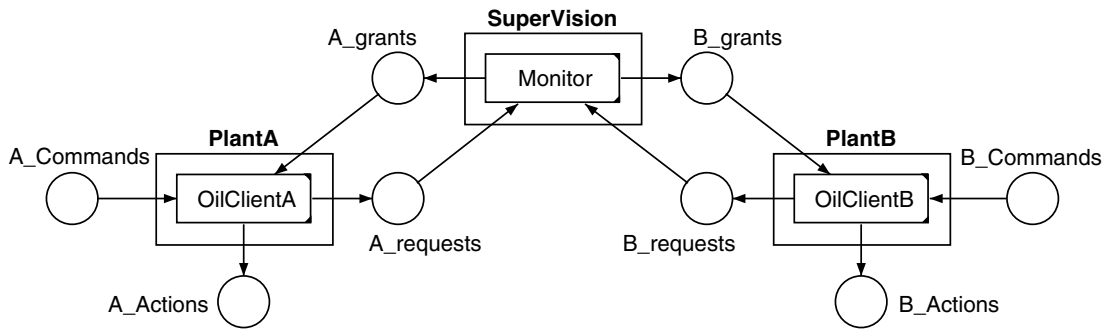
The solution *A* is asymmetric because the shared resource has not been modelled explicitly. In fact, the resource is implicitly modelled in the behavioural description of the *PlantA* process. The following generic resource access solution is based on a resource access model that is concurrent to both clients.

The specified model consists of three clusters. Nevertheless, the implementation of the model can also run on two processors only. We may for instance decide, to implement the *SuperVision* and the *PlantA* cluster on a single processor. The remaining difference to solution *A* is that pipeline access and exclusive resource access management is solved within different functional components.

## 6 The CIP Modelling Framework

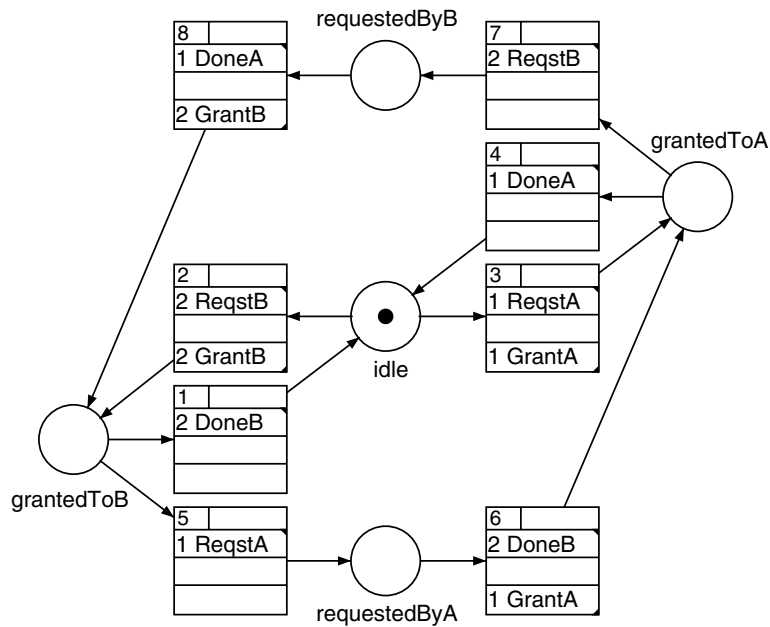
**SYSTEM** PipelineAccessControl\_MonitorSolution

COMMUNICATION NET



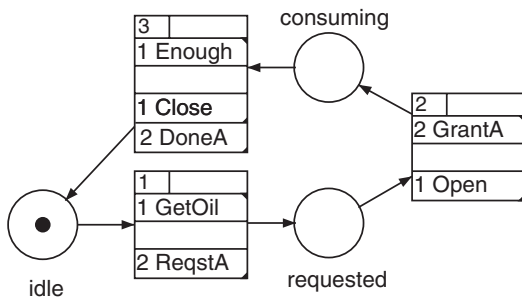
**CLUSTER** SuperVision

**PROCESS** Monitor



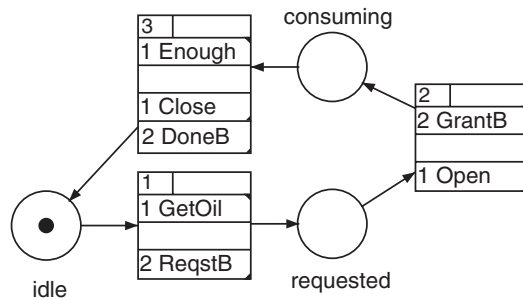
**CLUSTER** PlantA

**PROCESS** ClientA



**CLUSTER** PlantB

**PROCESS** ClientB



## 6.4 Pulse Cast Interaction

Pulse cast interaction is used to cause stimuli transmission between the processes of a cluster. Pulse casting is always initiated by process that has been triggered externally by an input message. An initiated chain of process reactions is always executed to its end before a further process can be triggered externally by an input message (run-to-completion semantics).

### 6.4.1 The Elements of a Pulse Cast Connection

A pulse cast is a multicast, performed as stimuli transmission from one to several processes of a cluster. Every process can act as sender of one, and react as receiver of several pulse cast connections. Pulse cast connections are created graphically in a PULSE CAST NET diagram (figure 38). A pulse cast connection is in general a multi-cast connection that is represented by a set of one-receiver connections with common sender. The set of OUTPUTS and the set of INPUTS of a process represent the output and the input interface dedicated to pulse cast interaction.

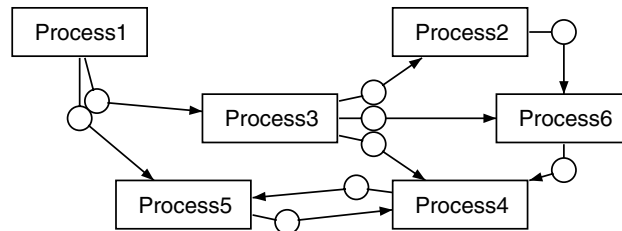


Figure 38: PULSE CAST NET – Processes linked with pulse cast connectors

The pulse cast net defines how processes are linked with pulse cast connectors. In order to specify the interaction of a pulse cast connection, the outputs of a sender must be mapped to inputs of the receivers. This mapping is termed PULSE TRANSLATION. In addition, for each pulse cast connection a CAST ORDER is specified which defines the sequential order of receiver activation when a pulse cast occurs.

The frame of Figure 39 shows all elements of a pulse cast connection. The LINKED INTERFACES of a connection consist of the set of outputs of the sender and set of inputs of the receivers. The CAST ORDER is defined as ordered receiver list.

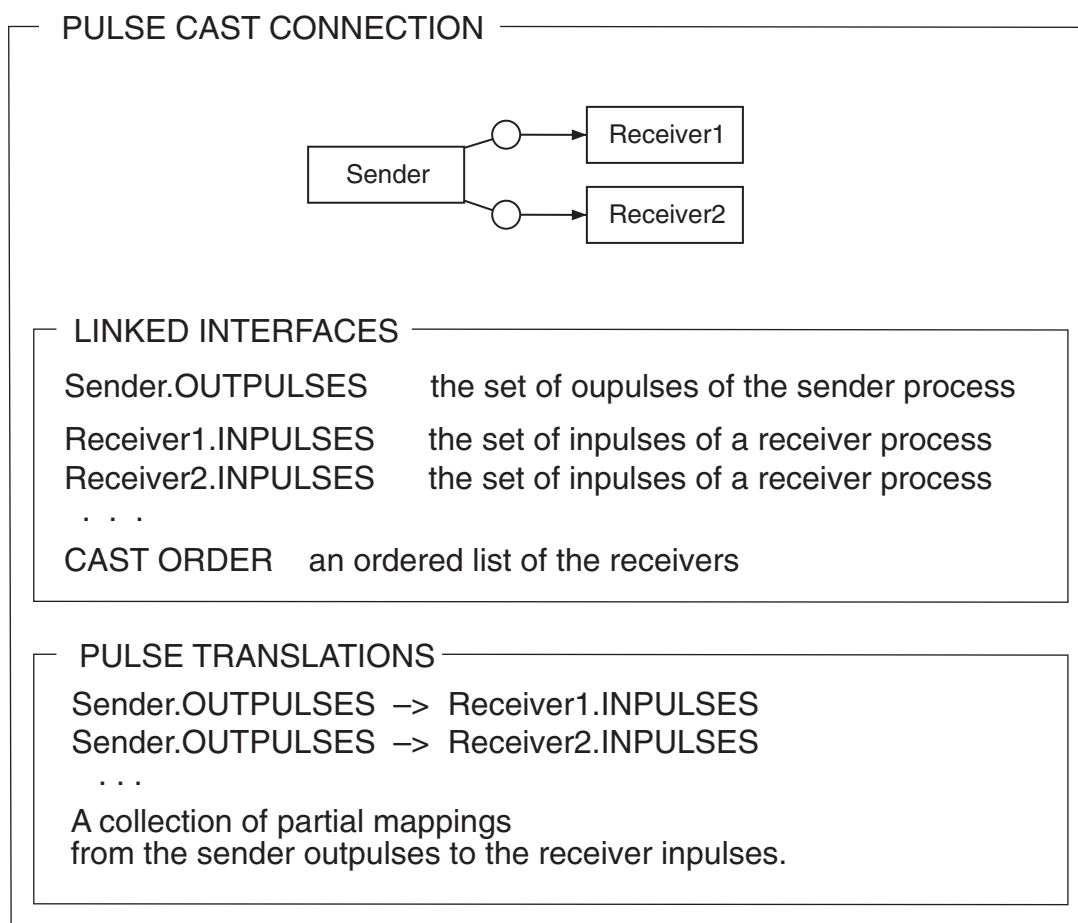


Figure 39: Pulse cast connection elements

The set of PULSE TRANSLATIONS represents the interaction relation which specifies the interaction mechanism of the particular pulse cast connection. For each sender-receiver pair of a pulse cast connection the pulse translation is defined as *partial mapping* from the sender outpulses to the receiver inpulses. The mapping is *partial* because a part only of the sender outpulses must be translated to the inpulses of a particular receiver. An outpulse that is not translated to a particular receiver, is not transmitted to that receiver when a multi-cast of that outpulse takes place.

A pulse translation between a sender and a receiver is a functional relation; that is, a particular outpulse can be mapped to one receiver inpulse only, but several outpulses can be mapped to the same inpulse. However, on the cluster level, the same outpulse is in general mapped to inpulses of several receivers (multi-cast).

### *Extended Pulse Cast Interaction.*

Plain pulse cast interaction specifies multicast between processes that represent plain state machines. In order to support pulse casting between extended state

machines, pulses can be extended, similarly as port messages, by data types of a programming language. Data type compatibility between sent and received pulses must be ensured when outpulses are mapped to inpulses: the data type of the outpulse and the associated inpulse must be equal. However, a typed outpulse can also be mapped to an untyped inpulse (no data transmission).

### 6.4.2 The Pulse Cast Interaction Mechanism

As a cluster is a synchronous composition of processes, the semantics of pulse cast interaction is based on the notion of synchronous state machine composition, introduced in section 5.2.

When a synchronous state machine composition is triggered by a stimulus, all state machines of the composition are involved simultaneously in a common reaction step; that is, some of the state machines perform a state transition while the other ones remain idle.

#### *Sequential Multicast*

There are two types of multicast mechanisms conceivable: *synchronous* cast and *sequential* cast. A *synchronous* multicast triggers the receivers synchronously, that is, the reaction of the receivers is performed simultaneously within one single reaction step of the cluster. A *sequential* multicast, on the other hand, causes a sequence of single-process cluster state transitions. In a single-process cluster state transition exactly one process performs a state transition while the other processes remain idle.

#### ***Sequential Multicast***

Pulse cast interaction is a *sequential* multicast. The sequential order of receiver activation is defined for each pulse cast connection by the specified cast order of receivers.

#### *Run-to Completion Semantics of Pulse Cast Interaction*

In addition, pulse cast interaction satisfies the behavioural restrictions imposed by run-to-completion semantics:

#### **Run-to-Completion Semantics**

As long as a cluster is to be triggered by generated pulses, external stimulation by inport messages is disabled.

As discussed in section 5.2, the idea of run-to-completion semantics suggest using an extensional behavioural description of synchronous state machine compositions by means of macro steps. Each macro step can be described in more detail by chains of micro steps. We adapt this terminology to cluster models:

### **process step (micro step)**

A *process step* is a single-process state transition of a cluster, caused by message passing or pulse cast interaction. During a process step, state inspection and mode control interaction may be active.

We often use verbalism like “a process is triggered by a pulse“. The exact meaning of such an expression is “the cluster performs a process step in which this process is activated by that pulse”.

### **cluster step (macro step)**

A *cluster step* is a complete sequence of process steps. During a cluster step, the cluster activity consists of performing process steps and pulse cast interaction. The first process step is triggered by a message passed by a channel. The subsequent process steps are triggered by impulses caused by pulse cast interaction. The cluster step terminates when all pulse cast interactions have been performed. Note that within a macro step, a particular process may in general be activated more than once. Due the run-to-completion semantics of cluster steps, the history of a cluster can be described by a unique sequence of cluster steps.

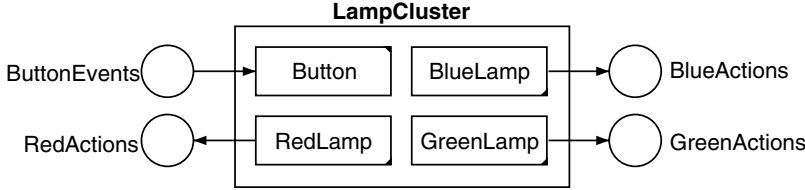
Two questions arise immediately. Is the sequence of process steps within a cluster step uniquely determined, and how is the termination of such a sequence of process steps ensured? The answer to these questions is given below in the subsection about execution rules of pulse cast chains. The deterministic execution of process steps is enforced by strengthening the execution rule of pulse cast interaction chains, and bounded cluster steps can be ensured by prohibiting recursive pulse cast interaction.

### **6.4.3 A Simple Pulse Cast Modelling Example**

The purpose of the simple example is to illustrate the multicast interaction mechanism of a single pulse cast connection. The CIP model consists of a single cluster. A *Button* process receives event notifications from a real button. A pulse cast connection is used to trigger tree lamp processes which control a blue, green and red lamp of the system environment.

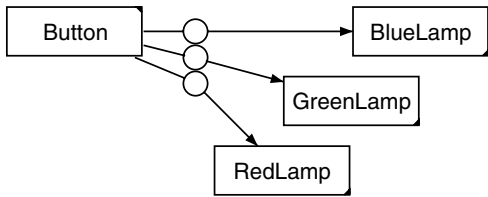
**SYSTEM** SimplePulseCast

COMMUNICATION NET



**CLUSTER** LampCluster

PULSE CAST NET

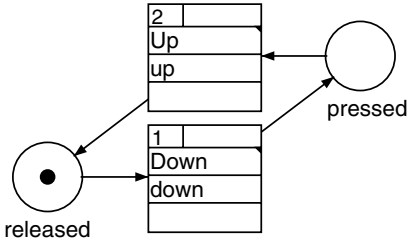


PULSE TRANSLATIONS

SENDER Button  
 down -> BlueLamp.on  
          -> GreenLamp.change,  
          -> RedLamp.change,  
 up     -> BlueLamp.off,  
          -> GreenLamp.change

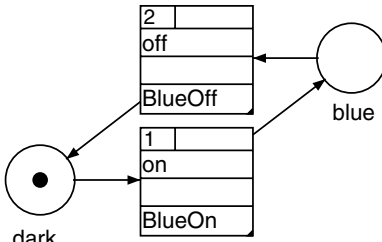
**PROCESS** Button

INPORT EventPort  
 MESSAGES Down, Up  
 OUTPUTS down, up  
 MODE normal



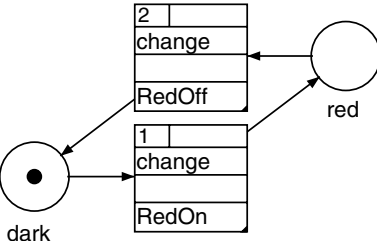
**PROCESS** BlueLamp

INPUTS off, on  
 OUTPORT ActionPort  
 MESSAGES BlueOff, BlueOn  
 MODE normal



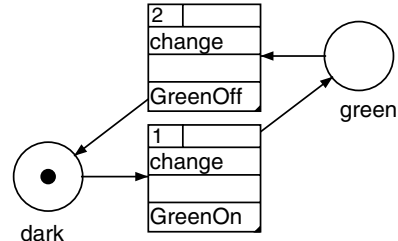
**PROCESS** RedLamp

INPUTS change  
 OUTPORT ActionPort  
 MESSAGES RedOff, RedOn  
 MODE normal



**PROCESS** GreenLamp

INPUTS change  
 OUTPORT ActionPort  
 MESSAGES GreenOff, GreenOn  
 MODE normal



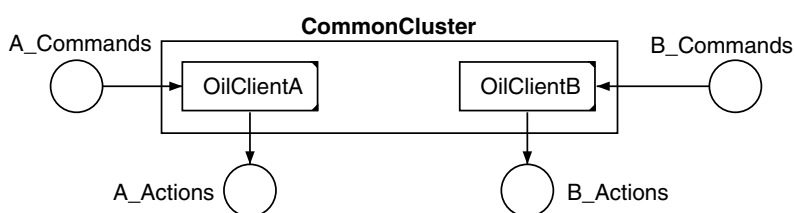
Each of the three pulse translations from the *Button* to the lamp processes is of a different type: *one-to-one*, *all-to-one* and *partial one-to-one*. Thus the blue and the red lamp are turned off and on when the button is pressed and released whereas the red lamp changes its state when the button is pressed only.

### 6.4.4 The Pulse Cast Solution of the PipelineAccessControl Problem

The incorrect solution *B* of the *PipelineAccessControl* problem (see 6.3.3) runs correctly if it is implemented on a single processor, and if the internal channels have a higher priority than the source channels. However, for a single processor implementation we can easily rectify the incorrect solution by placing both oil client processes into the same cluster, and by basing exclusive resource access control on run-to-completion semantics of pulse cast interaction.

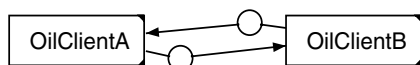
**SYSTEM** PipelineAccessControl\_PulseCastSolution

COMMUNICATION NET



**CLUSTER** CommonCluster

PULSE CAST NET



PULSE TRANSLATIONS

SENDER OilClientA

doneA -> OilClientB.doneA

userA -> OilClientB.userA

SENDER OilClientB

doneB -> OilClientA.doneB

userB -> OilClientA.userB

**PROCESS** OilClientA

INPORT cmd

MESSAGES Enough, GetOil

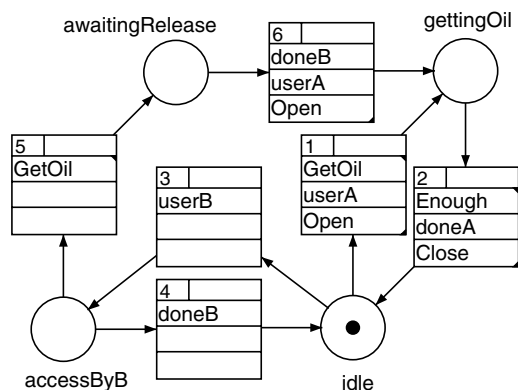
IMPULSES doneB, userB

OUTPULSES doneA, userA

OUTPORT act

MESSAGES Close, Open

MODE normal



**PROCESS** OilClientB

INPORT cmd

MESSAGES Enough, GetOil

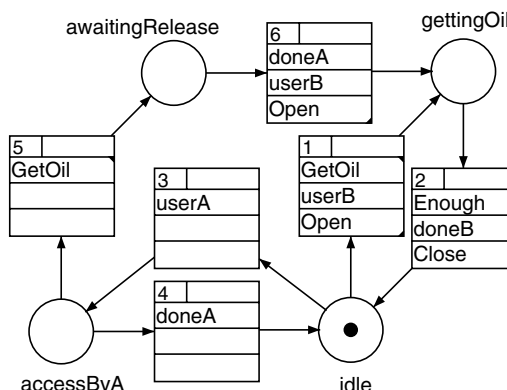
IMPULSES doneA, userA

OUTPULSES doneA, userA

OUTPORT act

MESSAGES Close, Open

MODE normal



### 6.4.5 Execution Rules for Pulse Cast Chains

As already mentioned above, the process steps of a cluster step must be executed in a deterministic way, and the maximal number of potential process steps within a cluster step must be bounded. Specifying deterministic execution of cooperating behavioural entities is a fundamental difficulty in the development of complex control systems. However, most modelling frameworks used in practice do not allow the user to specify deterministic cooperation properly. For instance, *Statecharts* allows, similarly to pulse cast interaction, specification of event multicast between orthogonal subcharts of a model, but the execution of the multicast is in general non-deterministic. Another example is *SDL*, a modelling framework based on asynchronous message passing between concurrent process. Because all messages are passed asynchronously, the execution of a model is strongly non-deterministic. Deterministic execution is often enforced partially – similarly to solution *B* of the *PipelineAccessControl* problem – by defining priorities when the individual message passing connections are implemented. In addition, neither *Statecharts* nor *SDL* support a semantic means to prevent unbounded system reactions.

The difficulty of specifying deterministic cooperation arises because the internal interaction propagation between cooperating entities is not linear. An external input to a system typically leads to branched chains of component reactions. In general, such chain reactions must even be expected to be cyclic. Thus deterministic execution of complex interaction chains can obviously not be based on local interaction mechanism only. There is a need of execution rules that suitably restrict the general behaviour resulting from the specified interaction mechanisms. An example of such a rule is the run-to-completion semantics of cluster steps. This rule prevents the interleaved execution of different cluster steps.

In the following we introduce an execution rule that strengthens the execution rule of pulse cast interaction chains in order to ensure deterministic cluster steps. The problem of unbounded cluster steps will be circumvented by requiring in addition non-recursive pulse cast interaction.

#### *Sequential Order of Process Steps: Run-to-Completion of Chain Reactions*

The cast order specified for the individual pulse cast connections of a cluster does in general not enforce a unique sequential order of the process steps occurring in a cluster step. The following pulse cast scenario is an example that allows executing the process steps in different orders. The scenario is visualized by the pulse cast tree depicted in Figure 40. Each graphic node of the tree structure represents a potential process step to be executed within the caused

## 6 The CIP Modelling Framework

cluster step. The node inscription denotes a process. The symbols above the graphic node denote the input stimuli of the individual process steps: the inport message  $m_1$  triggers process  $P$  and the inpulses  $ip_1, ip_2, \dots$  activate the other processes. The symbols  $op_1, op_2, \dots$  beneath a graphic node denote the outpulse generated by the process steps. Message outputs are not shown in the figure. The pulse transmission between process steps are represented by lines with arrows. The labelled tree structure abstracts from the process states leaved and reached in the executed process steps.

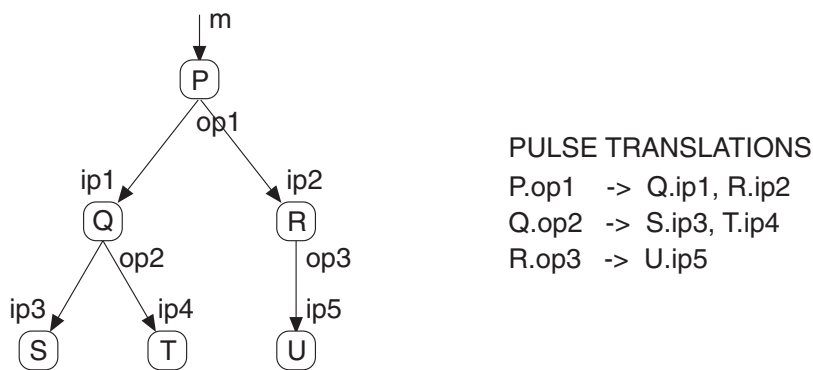


Figure 40: a pulse cast tree

The vertical tree order shows the *causal order* of the process steps. For instance, the steps of process  $S$  and  $T$  are caused by the step of process  $Q$ . The causal order of process steps follows directly from the connective structure of the pulse cast net. The execution order obviously respects the causal order; but in general, a causal order is a partial order only. For instance, the activation order of  $S$  and  $T$  is not specified by the pulse cast net.

The horizontal left-to-right order of the pulse cast tree represents the *cast orders* of the individual pulse cast connections. For instance, we can deduce from the tree structure that a multicast caused by process  $Q$  activates process  $S$  before process  $T$ .

However, the causal and the cast order are in general not sufficient to determine the execution order of the process steps uniquely. Let us give three possible sequences, called *deep-first*, *right-first* and *irregular*, that respect the causal order as well as the cast order:

$P, Q, S, T, R, U$  (deep-first)

$P, Q, R, S, T, U$  (right-first)

$P, Q, S, R, U, T$  (irregular)

Each of the three cluster steps performs a correctly ordered execution of process steps.

*When does the Execution Order of Process Steps Matter?* In our simple cluster step example each of the tree proposed execution sequences leads to the same cluster step; that is, the resulting cluster state and the set of generated output messages (not shown in the figure) are identical in all three execution cases.

Even in real projects, most of the potential cluster steps do not depend on the execution order of the involved processes. However, if we replace, for instance, in the pulse cast tree the step of process T by a further step of process R, the final state of the process R and even its message output can depend on the activation order of the two steps of process R.

Furthermore, if state inspection and mode control interaction takes place between processes that are activated in parallel subtrees, the result of the evaluated state dependencies can also depend on the order of process step execution. (State inspection and mode control interaction will be treated in section 6.5 and 6.5.)

*Run-to-Completion of Chain Reactions.* In order to ensure a deterministic execution of cluster steps we must claim an execution rule that determines the process step sequences uniquely:

**Run-to-Completion Semantics of Chain Reactions (deep-first)**

The chain reaction initiated by a receiver that has been triggered within a multicast is executed completely before the next receiver of that multicast is triggered.

Applying the rule to our example, we obtain the *deep-first* sequence as unique execution order. The execution rule yields for every potential cluster step a unique sequence of process steps. In mathematical terms the resulting process step sequence is a *deep-first traverse* of the right-to left ordered pulse cast tree.

The above introduced run-to-completion semantics of cluster steps appears now as a consequence of the stronger run-to-completion semantics of chain reactions: a cluster step is the chain reaction caused by the receiving of an external message.

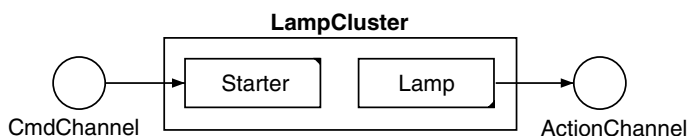
## 6 The CIP Modelling Framework

### *Bounded Cluster Steps – Non-Recursive Pulse Cast Interaction*

The notion of cluster step run-to-completion semantics implicitly supposes that the an externally caused pulse cast interaction terminates after a finite number of process steps. However, by defining cyclic pulse cast interaction structures it is possible to create models that yield the same pulse transmission chain repeatedly during a particular cluster step. The pulse cast tree of such a cluster step becomes an infinite tree because it contains the same reaction path an unbounded number of times. Figure 41 shows a simple model with a *recursive* pulse cast interaction.

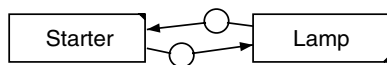
#### **SYSTEM** RecursivePulseCast

##### COMMUNICATION NET



#### **CLUSTER** LampCluster

##### PULSE CAST NET



##### PULSE TRANSLATIONS

SENDER Button  
change -> Lamp.change  
SENDER Lamp  
ack -> Button.ack

#### **PROCESS** Button

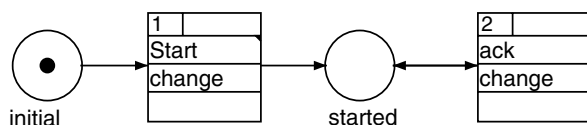
INPORT Commands

MESSAGES Start

INPULSES ack

OUTPULSES change

MODE normal



#### **PROCESS** Lamp

INPULSES change

OUTPULSES ack

OUTPORT Actions

MESSAGES Bright, Dark

MODE normal

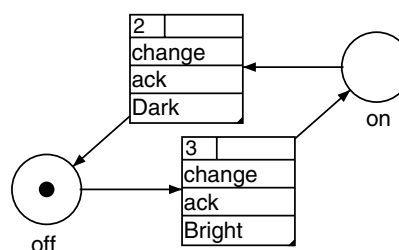


Figure 41: Example of recursive pulse-cast interaction

When the *Start* message occurs, the *Starter* process triggers with the *change* out-pulse a first *change* reaction of the *Lamp* process. The activated *Lamp* process acknowledges the caused *change* inpulse with an *ack* outpulse. Because the sub-

sequent *Starter* reaction on the received *ack* impulse causes by the emitted *change* outputpulse again a *change* impulse, an infinite repetition of *changelack* -> *ack/change* cycles occurs. Such a behaviour is termed a *live-lock*.

The repeated *changelack* -> *ack/change* cycle of our example represents a recursive pulse cast path because the reaction of the *Lamp* process on the *change* impulse causes a pulse propagation that activates the *Lamp* process again with the *change* impulse. In order to prevent unbounded interaction chains we must prohibit recursive pulse cast interaction.

***Enforcing Bounded Cluster Steps.*** In order to prevent unbounded interaction chains we prohibit recursive pulse cast interaction:

***Prohibition of Recursive Pulse Cast Interaction***

Within a cyclic activation of a process, the process must not be activated more than once by the same impulse.

If pulse cast interaction is non-recursive, all cluster steps are bounded because the number of process impulses of a CIP model is finite. However, in complex systems it is often difficult to locate recursive interaction paths. The effectively possible sequences of process steps depend not only on the specified pulse cast interaction, but also on the particular behaviour of each process. For instance, it is possible that the pulse cast interaction and the input-output relationships of the individual processes of a model permit repeated activation of a particular process by the same impulse; nonetheless the occurrence of unbounded cluster steps may be prevented through the local state dependency of reactions of the involved processes. Note, for instance, that a process can be in a state that does not expect a transmitted impulse, which can cause the termination of a cluster step.

***Automated Prevention of Recursive Pulse Cast Interaction.*** CIP Tool prevents the user from constructing CIP models with recursive pulse cast interaction.

***CIP Tool – Ensuring Non-Recursive Pulse Cast Interaction***

If a modelling action would lead to a cyclic *pulse propagation path* the modelling action is aborted.

A *pulse propagation path* describes how cast outputpulses cause transmitted impulses, and how activated processes can react on impulses by generating outputpulses. The notion of *pulse propagation path* abstracts from the state dependency of the output generation. Thus a pulse propagation path represents a whole class of potential cluster steps.

Because a pulse propagation path ignores process state, the condition of preventing cyclic pulse propagation paths is, really, over-restrictive for the purpose of guaranteeing bounded cluster steps. Interaction in a cluster with a cyclic pulse propagation path might be bounded, as mentioned above, because of particular state dependencies of process transitions. However, the simple restriction has proven to be a good engineering choice, because it enormously simplifies the analysis of interaction and the representation of interaction sequences. Experience in many industrial projects has shown that pulse propagation path acyclicity is not a severe restriction in real systems.

Furthermore, the restriction to acyclic pulse propagation paths allowed to automate pulse cast interaction analysis. An Interaction Tree Viewer of CIP Tool shows during modelling all potential pulse cast trees.

*Remark.* The mathematical description of *pulse propagation paths* and the foundation of the automated interaction analysis of CIP Tool is described in {REF}. All notions are explained on a simple CIP model.

*Remark.* A pulse propagation path corresponds to a *collaboration diagram* of UML. However, there is big difference in the meaning of the two graphical notations. A *pulse propagation path* visualizes a dynamic property of the CIP model under construction whereas a *collaboration diagram* represents a required interaction scenario that has to be verified when the system has been built.

### 6.4.6 A Pulse Cast Modelling Example with Interaction Chains

The purpose of the modelling example is to illustrate the pulse cast interaction mechanism when pulse cast interaction chains occur.

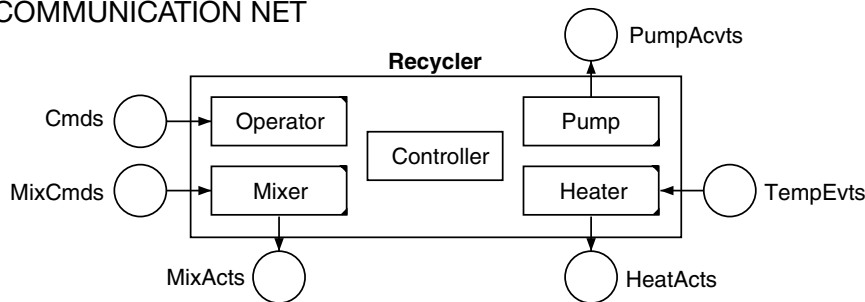
The *ChainedPulseCast* system models the control of a device which recycles a catalytic liquid that is used in chemical production process. The recycler embodies a heater, a mixer, and pump which causes a cyclic liquid exchange with the chemical production process.

When the recycler is started by the operator, the heater is to be turned on by the system. The pump must not be activated until the liquid in the recycler has reached the working temperature  $T$ . In addition, the mixer must have been turned on by the operator.

The *ChainedPulseCast* model consists of the single *Recycler* cluster only. The *Operator*, *Mixer*, *Heater* and *Pump* process represent context agents that are connected to the external processes of the embedded system. Internal interaction between context agents always takes place via the *Controller* process.

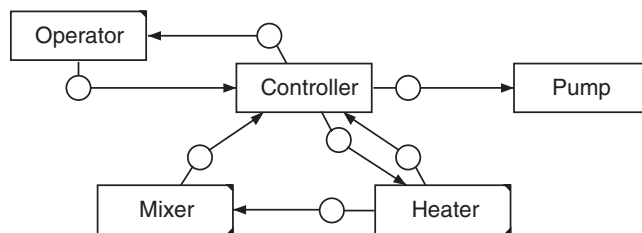
### SYSTEM ChainedPulseCast

#### COMMUNICATION NET



### CLUSTER Recycler

#### PULSE CAST NET



A *many-receiver* pulse cast connection is represented by set of *one-receiver* connections. For instance, the multi cast connection from the *Controller* to the *Operator*, *Heater* and *Pump* process consists of tree one-receiver connections.

#### CAST ORDER

SENDER Controller

RECEIVERS Heater, Pump, Operator

SENDER Heater

RECEIVERS Controller, Mixer

#### PULSE TRANSLATIONS

SENDER Controller

start -> Heater.start

stop -> Heater.stop, Pump.stop

work -> Pump.work, Operator.work

SENDER Heater

heatAck -> Controller.heatAck, Mixer.heatAck

SENDER Mixer

mixAck -> Controller.mixAck

SENDER Operator

begin -> Controller.begin

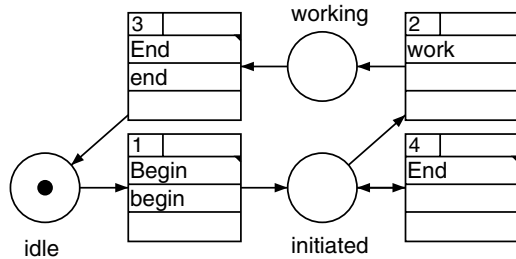
end -> Controller.end

Inpulses and associated outpulses have been given the same name. This naming style supports the readability of state transition diagrams of interacting processes. It will be applied in the following sample models also.

## 6 The CIP Modelling Framework

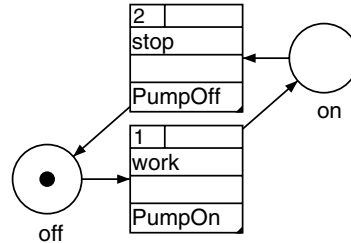
### PROCESS Operator

INPORT OpCmds  
 MESSAGES Begin, End  
 IMPULSES work  
 OUTPULSES begin, end  
 MODE normal



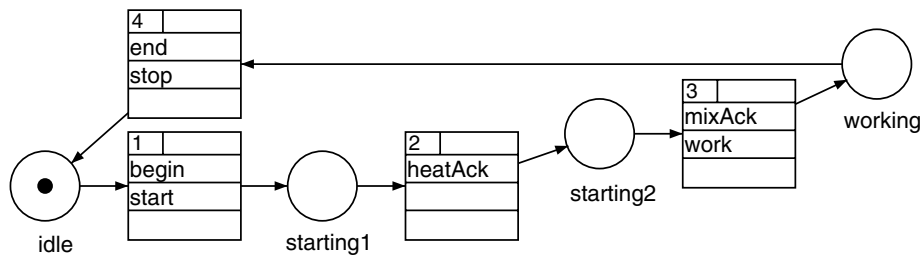
### PROCESS Pump

IMPULSES stop, work  
 OUTPORT PumpActions  
 MESSAGES PumpOff, PumpOn  
 MODE normal



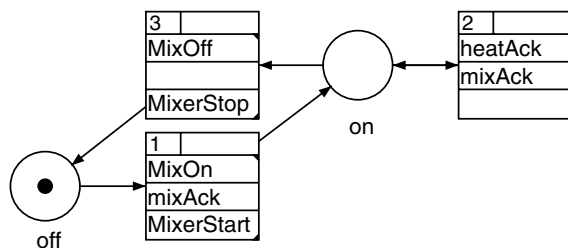
### PROCESS Controller

IMPULSES begin, end, heatAck, mixAck  
 OUTPULSES start, stop, work  
 MODE normal



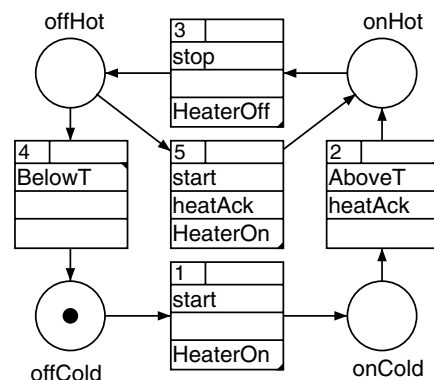
### PROCESS Mixer

INPORT MixCmds  
 MESSAGES MixOff, MixOn  
 IMPULSES heatAck  
 OUTPULSES mixAck  
 OUTPORT MixerActions  
 MESSAGES MixerStart, MixerStop  
 MODE normal



### PROCESS Heater

INPORT TempEvents  
 MESSAGES AboveT, BelowT  
 IMPULSES start, stop  
 OUTPULSES heatAck  
 OUTPORT HeaterActions  
 MESSAGES HeaterOff, HeaterOn  
 MODE normal



When the operator starts the recycler by entering the *Begin* message for the *Operator* process, the occurring pulse cast chain depends on the state of the *Mixer* and the *Heater* process.

If both processes are in the initial state, the caused cluster step is

```
Begin [Operator] begin •
  begin [Controller] start •
    start [Heater]
```

*Pulse Cast Tree Notation*  
inpulse [Process] outpulse  
*represents a process reaction*  
*(message output*  
*is not shown)*

Entering the *MixOn* message leads to

```
MixOn [Mixer] mixAck •
  mixAck [Controller] (pulse ignored)
```

```
outpulse •
  inpulse1
  inpulse2
represents a pulse cast
```

When the *AboveT* message notifies that the temperature *T* has been reached, the cluster step

```
AboveT [Heater] heatAck •
  heatAck [Controller]
  heatAck [Mixer] mixAck •
    mixAck [Controller] work •
      work [Pump]
      work [Operator]
```

finally causes the pump to be turned on.

The longest pulse cast chain initiated by the *Begin* message occurs when the mixer is turning already (*Mixer* state *on*), and the temperature has not yet been dropped below *T* (*Heater* state *offHot*). The pump is activated within a single cluster step caused by the *Begin* message.

```
Begin [Operator] begin •
  begin [Controller] start •
    start [Heater] heatAck •
      heatAck [Controller]
      heatAck [Mixer] mixAck •
        mixAck [Controller] work •
          work [Pump]
          work [Operator]
```

During this cluster step the *Controller* process is activated three, and the *Operator* process two times.

The state inspection interaction introduced in the next section will provide us a means, that allows replacing the used acknowledging technique (*ack* pulses) by explicitly specified state dependencies among cluster processes. The use of state inspection interaction leads in general to shorter pulse cast chains.

## 6.5 State Inspection Interaction

A *transition branching* of a process consists of several state transitions that await in the same state the same input message or impulse. State inspection interaction is used to establish dependencies on other process states in order to determine the execution of transition branchings. Furthermore, branching decisions can be based on conditions that can depend through procedure calls on the local variables of other cluster processes.

### 6.5.1 The Elements of a State Inspection Connection

A state inspection connection defines a true/false-valued function (boolean function) that depends on the states of one or more processes of the same cluster, and that updates a GATE (boolean interface state) of the inspecting process. State inspection connections are created graphically in an INSPECTION NET diagram. An example of an inspection net is shown in Figure 42.

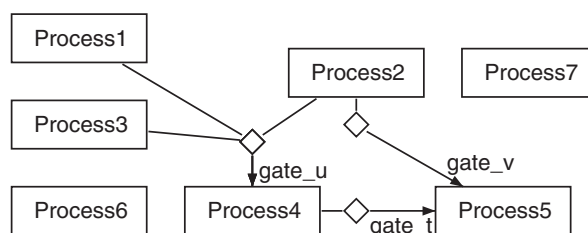


Figure 42: INSPECTION NET – Processes linked with inspection connectors

An inspection connector is linked to a number of inspected processes and to a GATE of the inspecting process. The arrow of an inspection connection indicates, as in stimulation connectors, the causal dependency of the connected processes. A particular gate can participate in one inspection connection only.

The frame of Figure 43 shows all elements of an inspection connection. The LINKED INTERFACES consist of a gate of the inspecting process and of the sets of states of the inspected processes. The boolean function defined by the TRUTH TABLE represents the interaction relation of the connection. The table defines a mapping, that associates to each tuple of inspected process states the value TRUE or FALSE.

The extension of state inspection interaction to extended state machines is called *state vector inspection*. The extended inspection mechanism is based on procedural process variable access and is described separately in section section 6.5.4.

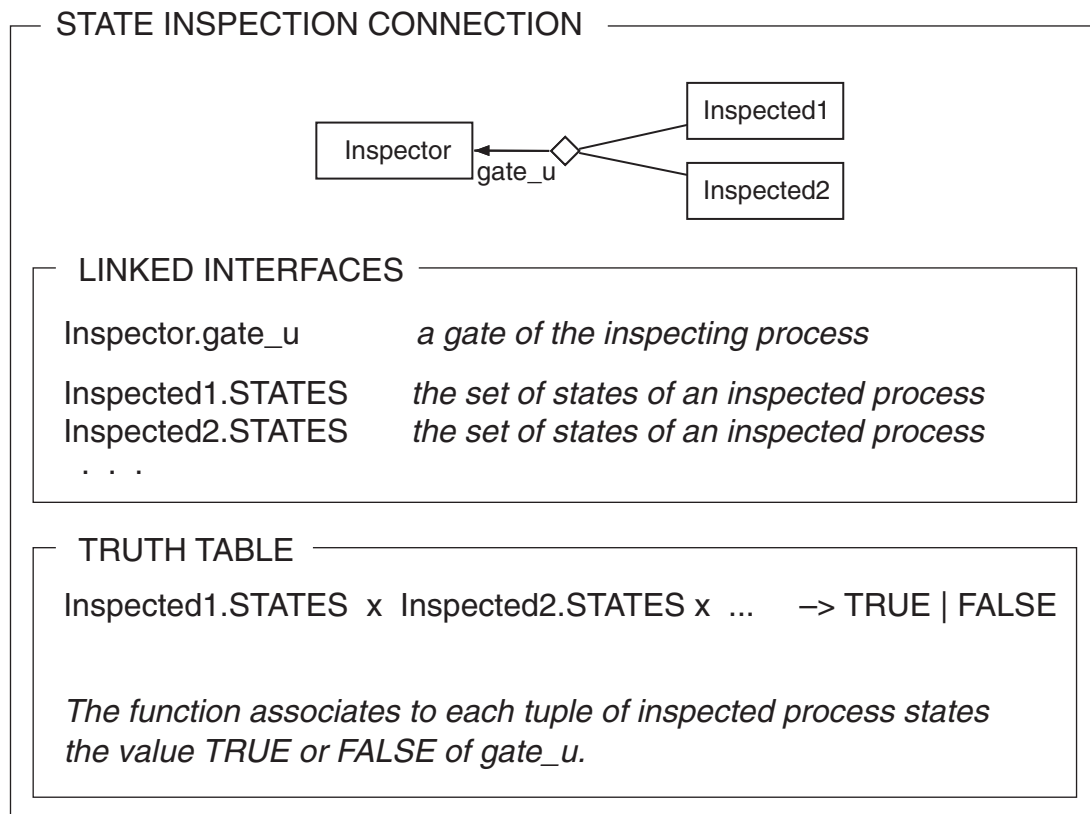


Figure 43: Elements of a state inspection connection

### 6.5.2 The State Inspection Interaction Mechanism

In contrast to stimulation by message passing and pulse cast, state inspection interaction is based on state sharing. Let us first reconsider the general mechanism of stimulation interaction. During a stimulation, at least two processes participate actively in the interaction. A stimulus is produced by a state transition of a sender process and triggers a state transition of one or more receiver processes. Stimuli transmission takes place *between* the active phases of the involved sender and receiver processes.

State inspection interaction is different. The interaction is initiated when the inspector of a connection has been activated by a stimulus that triggers a transition branching with associated gates (see section 6.2.2). The activated state inspection mechanism updates the gates associated to the branching transition. The gate evaluated to TRUE enables triggering the transition it is associated to. If more than one gate is evaluated to TRUE, the non-determinism of the branching is not completely resolved. However, specifying non-exclusive state inspection connections for the gates associated to a transition branching is considered as a specification error.

## 6 The CIP Modelling Framework

State inspection takes place when the *inspecting* process is active. The inspected processes remain inactive during an inspection, thus an inspection never has an effect on the state of the inspected processes.

### 6.5.3 A Simple State Inspection Modelling Example

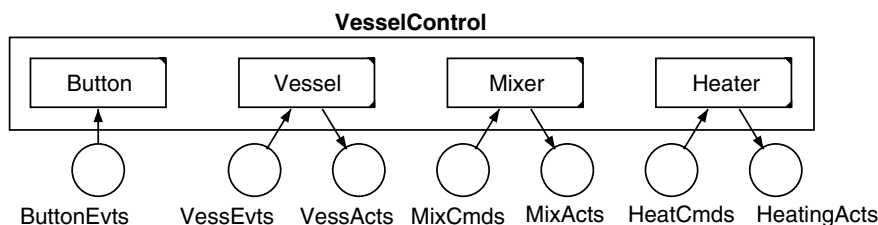
A vessel with an incorporated mixer and heater can be filled and emptied by pressing a button. The mixer and the heater are controlled by operator commands; the mixer can turn at low and at high speed. To empty the filled vessel, the heater must be turned off, and the mixer must not turn at high speed.

In the *SimpleStateInspection* model below, button press events are notified by the *Button* process to the *Vessel* process, which is responsible for filling and emptying the vessel. The *Mixer* and the *Heater* process control the mixer and the heater. For simplicity, these processes are modelled as processes that depend on operator commands only.

When the *Vessel* process is triggered to empty the vessel, the process reaction depends on the *Mixer* and *Heater* process state. This state dependency is modelled by a state inspection connection which updates the *enableDraining* gate.

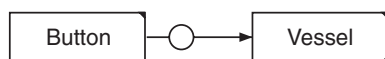
SYSTEM SimpleStateInspection

COMMUNICATION NET



CLUSTER LampCluster

PULSE CAST NET

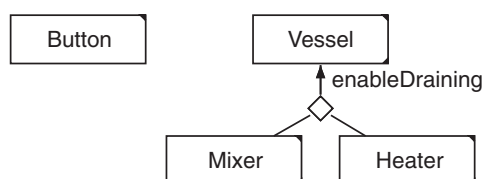


PULSE TRANSLATIONS

SENDER Button  
press -> Vessel.press



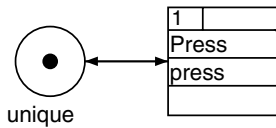
INSPECTION NET



TRUTH TABLES

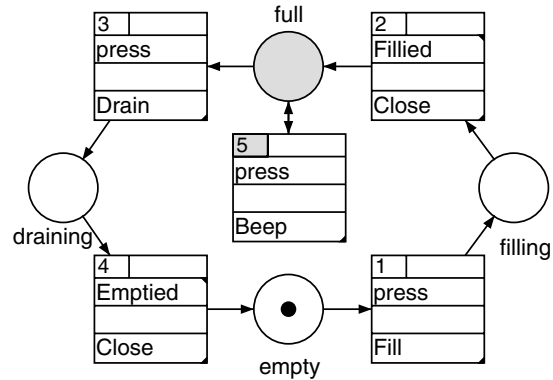
INSPECTOR Vessel  
GATE enableDraining  
RESPONDERS Mixer, Heater  
TRUE <- (mixerOff, heaterOff),  
(reduced, heaterOff)

**PROCESS** Button  
 INPORT EventPort  
 MESSAGES Press  
 OUTPULSES press  
 MODE normal



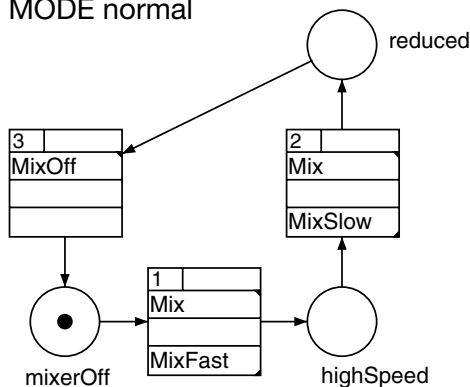
When the *Vessel* process is triggered in the *full* state by a *press* pulse, two state transitions are possible. When the *enableDraining* gate is true, the transition 3 to the *draining* state causes emptying the vessel; otherwise the transition 5 back to the *full* state causes a *Beep* signal.

**PROCESS** Vessel  
 INPULSES press  
 OUTPORT ActionPort  
 MESSAGES Close, Drain, Fill, Beep  
 GATE enableDraining  
 MODE normal

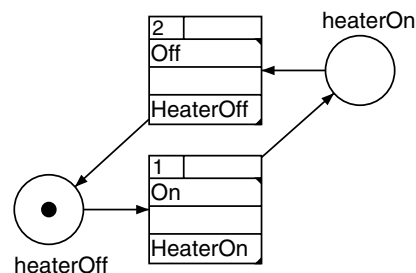


**SWITCH**  
 STATE full INPULSE press  
 TRANSITION 3 / enableDraining  
 TRANSITION 5 / ELSE\_

**PROCESS** Mixer  
 INPORT CmdPort  
 MESSAGES Mix, MixOff  
 OUTPORT ActionPort  
 MESSAGES MixFast, MixOff, MixSlow  
 MODE normal



**PROCESS** Heater  
 INPORT CmdPort  
 MESSAGES Off, On  
 OUTPORT ActionPort  
 MESSAGES HeaterOff, HeaterOn  
 MODE normal



The presented solution is not safe because the operator can turn the heater on or accelerate the mixer while the vessel is emptying. In order to disable such invalid actions we ought to extend the state transition structures of the *Heater* and the *Mixer* process by state transitions that branch to an *awaitingFillingVessel* state. The branch decisions can again be based on state inspection interaction, namely on inspection of the *Vessel* process state. The stipulated elaboration of the presented model is proposed as an exercise to the reader.

### 6.5.4 Extended State Inspection: State Vector Access by Queries

State inspection interaction models state dependencies on the discrete states of processes. In order to support the specification of dependencies on process variables (extended state machines) a procedural state vector access mechanism termed *state vector inspection* is introduced.

The frame of Figure 44 shows all elements of a state vector inspection connection. Connections are created also in the inspection net diagram of a cluster: the icon of the connector is marked in grey to indicate the extended inspection mechanism. A state vector inspection connector can be linked to one inspected process only. The two LINKED INTERFACES consist of the set of QUERIES of the inspecting process and the set of INQUIRIES of the inspected process. A query of a process is a function pointer (virtual function) described in the programming language of the state machine extension. An inquiry is an implemented state vector access function. The set of QUERY-INQUIRY LINKS of a connection is a mapping that defines how queries of the inspecting process are implemented by inquiries of the inspected process. The mapping must therefore relate inquiries to queries of the same function type. The same inquiry can be provided as implementation to queries of several inspecting processes, whereas each query must have a unique implementation defined by one particular inspection connection.

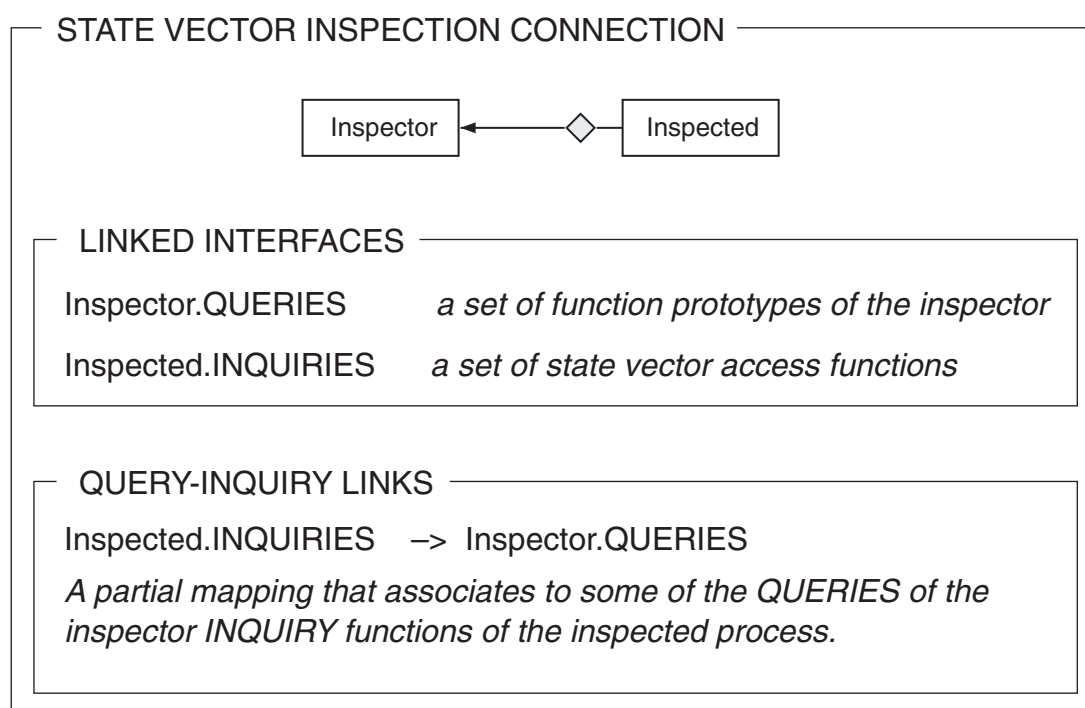


Figure 44: The elements of a state vector inspection connection

When an inspecting process calls a query the linked inquiry of the inspected process is invoked. Queries are called within conditions and operations. Within conditions associated to state transitions of a branching, query calls establish dependencies of process reactions on instance variables of other processes. Within operations, query calls can be used for instance in algorithms that depend on parameter values that are managed by an other process. Queries can be called also within the code of an inquiry (functional composition of services).

Using procedure calls to encapsulate state sharing interaction is well known from object-oriented programming languages. However, there is an essential difference between a query interface and an object interfaces. Queries represent interfaces that specify *required* services of a client, whereas object interfaces declare *provided* services of a server object. The link between query and inquiry defines the interaction between client and server whereas the link between object interface and local methods defines the implementation of the object interface. Interaction between client and server object is defined by means of service calls (method calls) placed in the code of the client object.

### 6.5.5 A Simple State Vector Inspection Modelling Example

The action parameters of a computer controlled mixer and a heater can be updated during run time by entering a corresponding configuration message.

In the *SimpleStateVectorInspection* model below, new parameter values can be entered to the *Config* process, which stores them in its state vector variables: the *RPM\_Mixer* variable holds the rotation rate per minute of the mixer, and the *temp\_tuned* variable holds the temperature level below which the heater is to be turned off. In addition, there is the *temp\_default* float constant specified which defines a default temperature level value. The default constant is defined in the INCLUDE text of the *Config* process.

The *Set Default* and the *Tuned* messages entered by the operator determine which temperature level must be used to control the heater.

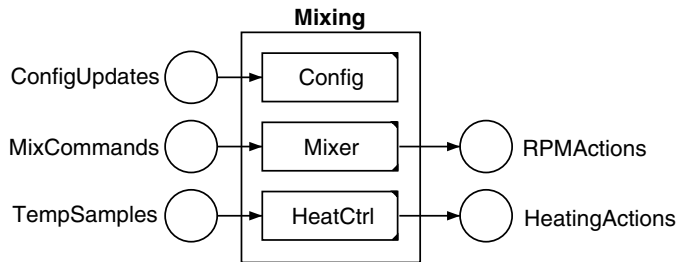
The current control parameters are provided to the *Mixer* and the *HeatCtrl* process by the corresponding *RPM\_Mixer* and *Temp\_Level* inquiry of the *Config* process.

These inquiries are invoked by query calls *getRPM()* and *getTempLevel()* issued by the *Mixer* and the *HeatCtrl* process. The necessary state vector connectors are defined in the PULSE CAST & INSPECTION NET of the *Mixing* cluster. The interaction is specified in the associated the QUERY-INQUIRY LINKS table.

## 6 The CIP Modelling Framework

SYSTEM SimpleStateVectorInspection

COMMUNICATION NET



USER TYPE tConfig

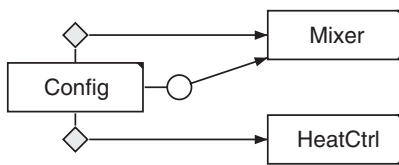
```
typedef struct{
    float newRPM;
    float newTempLevel;
}tConfig;
```

USER TYPE tFloat

```
typedef float Float;
```

CLUSTER Mixing

PULSE CAST & INSPECTION NET



PULSE TRANSLATIONS

SENDER Config  
change -> Mixer.change

QUERY-INQUIRY LINKS

Mixer.getRPM -> Config.RPM\_Mixer

HeatCtrl.getTempLevel -> Config.TempLevel

**PROCESS** Mixer

OUERY getRPM

```
float (*getRPM)(void)
```

INPORT CmdPort

MESSAGES MixOff, MixOn

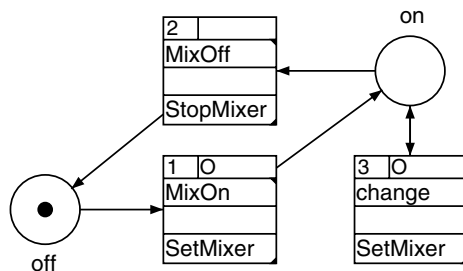
OUTPORT ActPort

MESSAGES SetMixer, StopMixer

OPERATION setRPM

```
{OUTMSG = getRPM();}
```

MODE normal



OPERATION ASSIGNMENT

TRANSITION 3 /

OUTPORT ActPort OPERATION setRPM

TRANSITION 4 /

OUTPORT ActPort OPERATION setRPM

**PROCESS** HeatCtrl

OUERY getTempLevel

```
float (*getTempLevel)(void)
```

INPORT Sampling

MESSAGES Sample: FLOAT

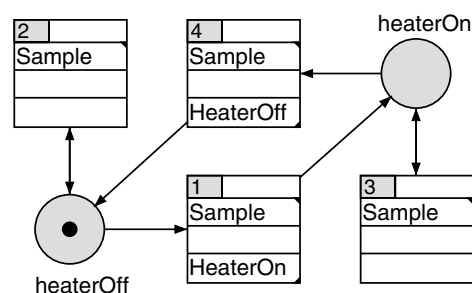
OUTPORT ActionPort

MESSAGES HeaterOff, HeaterOn

CONDITION tempLevelReached

```
IN > getTempLevel()
```

MODE normal



SWITCHES

STATE heaterOn MESSAGE Sample

TRANSITION 4 / tempLevelReached

TRANSITION 3 / ELSE\_

STATE heaterOff MESSAGE Sample

TRANSITION 2 / tempLevelReached

TRANSITION 1 / ELSE\_

**PROCESS** Config**INCLUDE**

```
#define temp_default 80
```

**INQUIRY** TempLevel

```
float TempLevel (void)
{
    if (STATUS.STATE == tuned)
        return STATUS.temp_tuned;
    else
        return temp_default;
}
```

**INQUIRY** RPM\_Mixer

```
float RPM_Mixer (void)
{
    return STATUS.RPM_Mixer;
}
```

**VARIABLES**

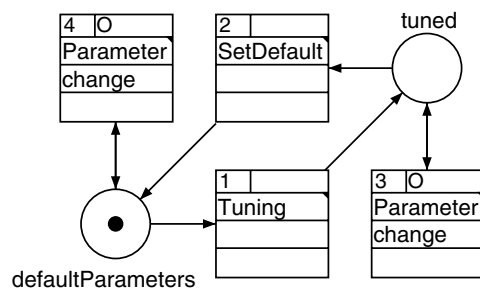
```
RPM_Mixer: tFloat, temp_tuned: tFloat
```

**INPORT** ConfigPort

```
MESSAGES Parameter: tConfig, SetDefault, Tuning
```

**OUTPUTSES** change**OPERATION** getNewConfig

```
{
    STATUS.RPM_Mixer = IN.newRPM;
    STATUS.temp_tuned = IN.newTempLevel;
}
```

**MODE** normal**OPERATION ASSIGNEMENT**

```
TRANSITION 3 / OPERATION getNewConfig
```

```
TRANSITION 4 / OPERATION getNewConfig
```

Note that queries can be called in any programming language construct (condition, operation, inquiry, ... ) of an inspecting process.

## 6.6 Mode Control Interaction

The full behaviour of a process can in general be specified by a number of alternative modes, each described graphically by a state transition diagrams. The active mode of process is selected by mode control interaction. The underlying interaction mechanism is based, like state inspection interaction, on state dependencies: the active mode of process is determined by the current states of one or more other processes of the same cluster.

### 6.6.1 Behavioural Structuring by Means of Process Modes

The state transition structure of a CIP process specifies how the process must react to inputs by state changes and generated outputs. Often such a behaviour becomes quite complex due to inputs which must influence the full future behaviour of the process. An alarm message, for instance, must cause a behaviour that differs from the normal case until the alarm is reset. The resulting transition structure will then represent a kind of superposition of the structures for the normal and the alarm case.

In order disentangle the superposed description of different behaviours, the full behaviour of a process is modelled by a number of alternative modes. Each mode is specified graphically by a corresponding state transition diagram, based on the states, message and pulse interfaces of the process. Figure 45 shows a door control process with a *userControl* and a *forcedClosing* mode. The *userControl* mode specifies the ordinary control of the door. The *forcedClosing* mode describes an alternative behaviour of the process, usable when an alarm or error condition occurs.

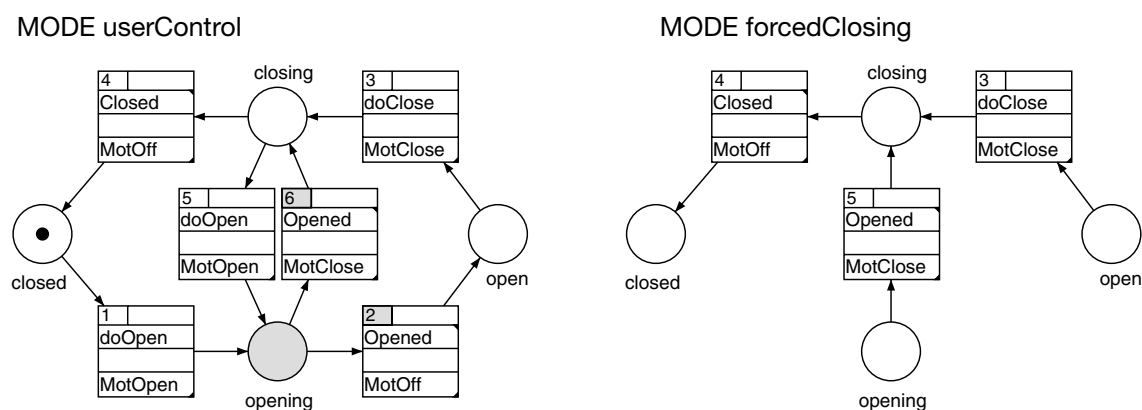


Figure 45: Two modes of a door control process

One mode only of a process can be active at the time. In order to render the choice of the active mode dynamically dependent on the behaviour of other

cluster processes, a kind of higher order interaction termed *mode control* is introduced.

### 6.6.2 The Elements of a Mode Control Connection

A mode control connection defines how the active mode of a process is determined by the current states of a number of processes of the same cluster. The controlled process of a mode control connection is termed *slave* while the controlling processes are termed *masters*. Mode control connections are created graphically in a hierarchical MODE CONTROL GRAPH, as shown in Figure 46. Mode control connectors are represented by triangles which are connected at the bottom angle to a slave and at the top side to the controlling masters. The graph is restricted to be acyclic in order to define a hierarchical structure; that is, each directed path formed by mode control connectors defines a finite master-slave order.

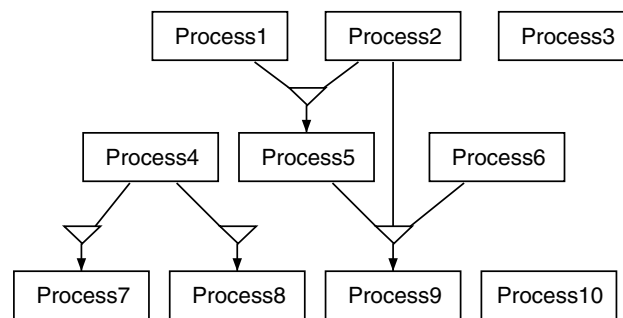


Figure 46: MODE CONTROL NET – Processes linked with mode control connectors

The levels indicated in the graph of Figure 46 do not define any behavioural restriction. Levelling is merely introduced informally to group processes that cooperate on a common level of abstraction. Usually not all processes of a cluster are involved in the hierarchical structure.

The frame of Figure 47 shows all elements of a mode control connection. The LINKED INTERFACES consist of the set of modes of the slave process and of the sets of states of the master processes of the connection. The MODE SETTING TABLE represents the interaction relation of the connection. It is defined as a function from the Cartesian product of master states to the set of modes of the slave. That is, to each tuple of master states there is a uniquely associated slave mode. If a slave has more than one master, the mode setting function expresses a kind of master agreement about the mode control of the shared slave.

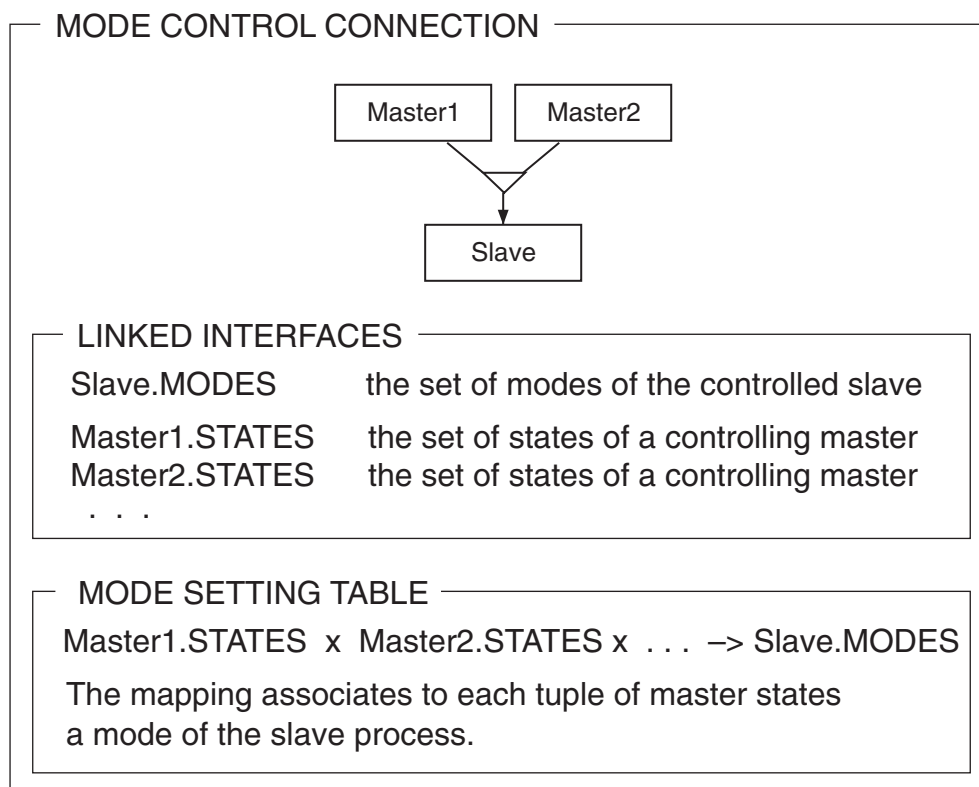


Figure 47: The elements of a mode control connection

### 6.6.3 The Mode Control Interaction Mechanism

The mode control interaction mechanism is similar to the interaction mechanism of state inspection. The initial mode of a slave process is determined by the initial states of the connected master processes. Mode control interaction is initiated when the slave process of a connection has been activated by a message or a pulse. The mode control connection evaluates the mode that defines the reaction to the occurred stimulus. Thus mode control, like state inspection, takes place when the dependent process is active.

Although each state transition of a master implicitly represents a potential mode change of the connected slave, the mode change becomes effective later on only, when the slave is triggered by one of its input stimuli. Suppose, for instance, that the door control process of Figure 45 is in the *closed* state of the *normal* mode when a master induces a change to the *shutting* mode. The mode change becomes sensible only to the door control process when the *open* pulse is sent to it. As a result of the induced mode change, the door will remain in the *closed* state.

It is important to note that a change of mode does not affect the current state of a slave; a state can change only when a transition in the active mode is trig-

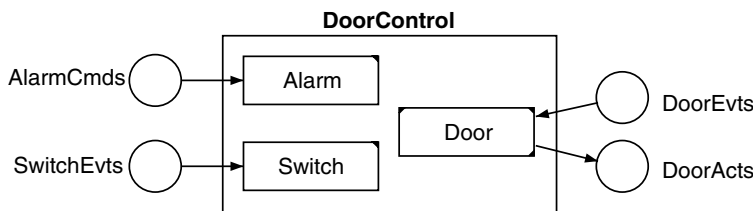
gered by a message or a pulse. The stability of state with respect to mode change is an essential property of mode control interaction. The current state of a process partially expresses the history of the occurred stimuli, and thus represents information about the behavioural context of the process. This context information is essential to determine reactions of the process when a stimulus occurs, and it becomes crucial that this information is not lost when a process changes its current behaviour.

### 6.6.4 A Simple Mode Control Modelling Example

A button is used to open and close a door. In addition, if an operator enters an alarm command, the door must be immediately closed. When the alarm is released by the operator, the door can again be opened and closed by the button.

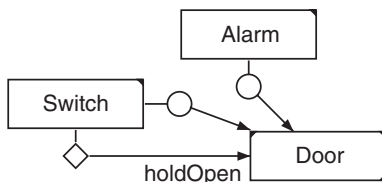
**SYSTEM** SimpleModeControl

COMMUNICATION NET



**CLUSTER** LampCluster

PULSE CAST & INSPECTION NET



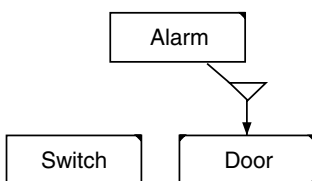
PULSE TRANSLATIONS

SENDER Alarm  
doClose -> Door.doClose  
SENDER Switch  
doClose -> Door.doClose  
doOpen -> Door.doOpen

TRUTH TABLE

INSPECTOR Door GATE holdOpen  
RESPONDERS Switch  
TRUE <- down

MODE CONTROL GRAPH



MODE SETTING

SLAVE Door  
MASTERS Alarm  
forcedClosing <- alarm  
userControl <- regular

## 6 The CIP Modelling Framework

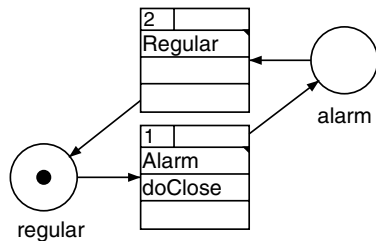
### PROCESS Alarm

INPORT CmdPort

MESSAGES Alarm, Regular

OUTPUTS doClose

MODE normal



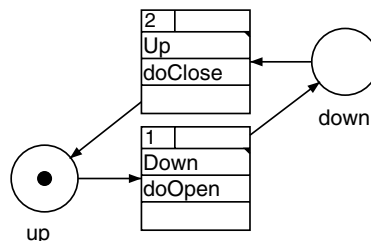
### PROCESS Switch

INPORT EventPort

MESSAGES Down, Up

OUTPUTS doClose, doOpen

MODE normal



### PROCESS Door

INPORT PositionEvents

MESSAGES Closed, Opened

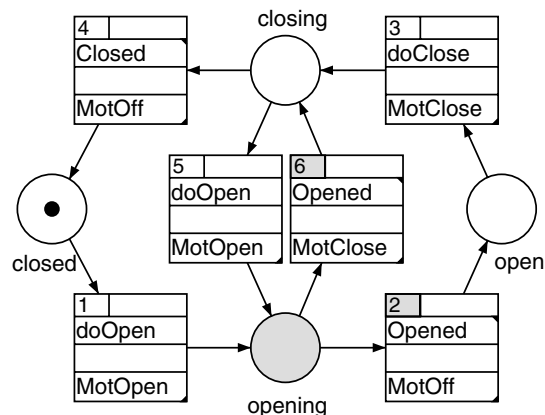
IMPULSES doClose, doOpen

OUTPORT MotorActions

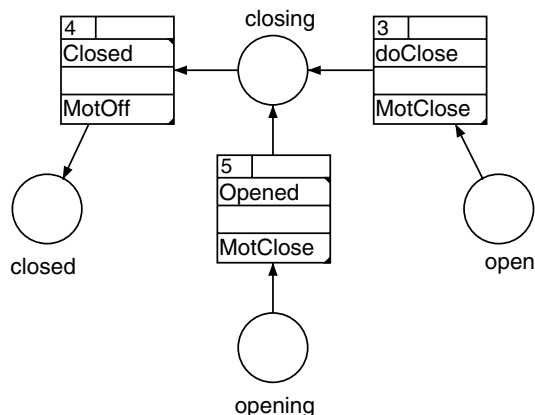
MESSAGES MotClose, MotOff, MotOpen

GATE holdOpen

MODE userControl



MODE forcedClosing



### SWITCH

STATE opening MESSAGE Opened

TRANSITION 3 / holdOpen

TRANSITION 6 / ELSE\_

When the *Alarm* process changes from the *regular* to the *alarm* state, the active mode of the *Door* process changes from *userControl* to *forcedClosing*. The emitted *doClose* pulse of the *Alarm* process is received by the *Door* process in the activated *forcedClosing* mode and triggers the process in the *opening* state to close the door. When the *Alarm* process switches back to the *regular* state, the active mode of the *Door* process is correspondingly changed to *userControl*.

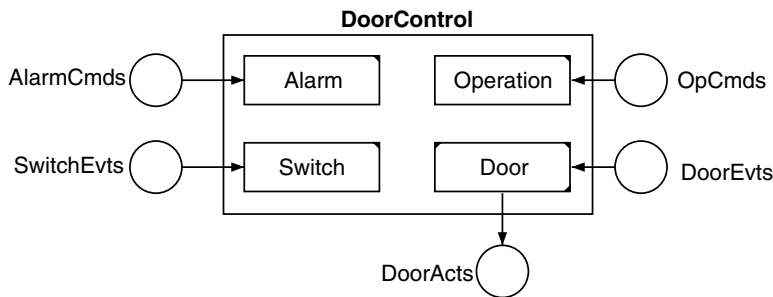
Note that an alarm may be released immediately after commanding the alarm. In this case it is possible that after releasing the alarm the button is pressed to reopen the door before the door has completely been closed.

6.6.5 Extended Mode Control Modelling Example

The ordinary control mode (*userControl*) of the door of the *SimpleModeControl* model is extended by two additional modes of operation: in a *scheduled* mode the door is opened and closed at determined points in time while in a *holdingOpen* mode the door must remain open. In addition, the operator can issue a *UserCtrl*, a *Schedule* and or a *HoldOpen* command to activate the corresponding door mode. However, when an *Alarm* occurs, the door still must be closed immediately. When the alarm is released the door is again to be controlled in the mode of operation set by the operator.

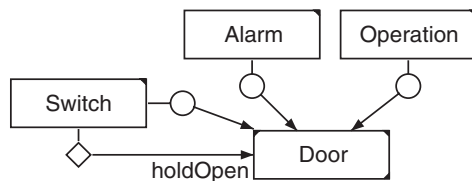
**SYSTEM** ExtendedModeControl

COMMUNICATION NET



**CLUSTER** LampCluster

PULSE CAST & INSPECTION NET



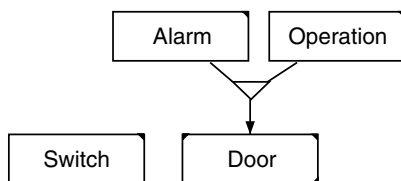
TRUTH TABLE

INSPECTOR Door GATE holdOpen  
 RESPONDER Switch  
 TRUE <- down

PULSE TRANSLATIONS

SENDER Alarm  
 doClose -> Door.doClose  
 initTimer -> Door.initTimer  
 SENDER Operation  
 doOpen -> Door.doOpen  
 initTimer -> Door.initTimer  
 SENDER Switch  
 doClose -> Door.doClose  
 doOpen -> Door.doOpen

MODE CONTROL GRAPH

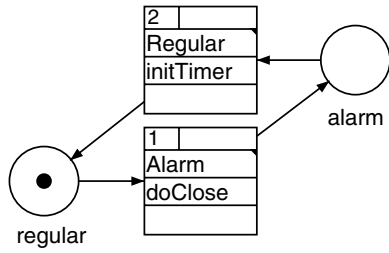


MODE SETTING

SLAVE Door  
 MASTERS Alarm, Operation  
 forcedClosing <- (alarm, openDoor),  
 (alarm, scheduled),  
 (alarm, userControlled)  
 holdingOpen <- (regular, openDoor)  
 scheduled <- (regular, scheduled)  
 userControl <- (regular, userControlled)

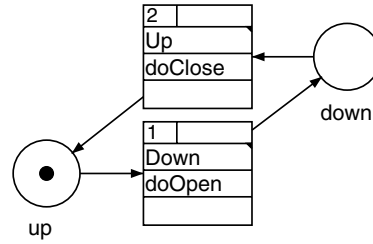
# 6 The CIP Modelling Framework

## PROCESS Alarm



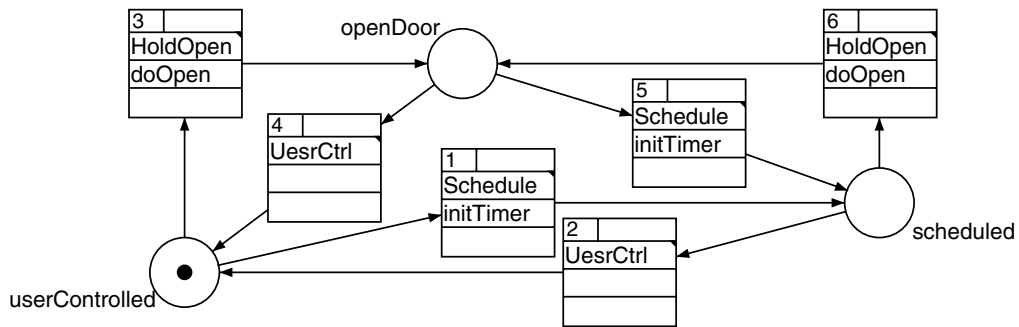
## PROCESS Switch

MODE normal



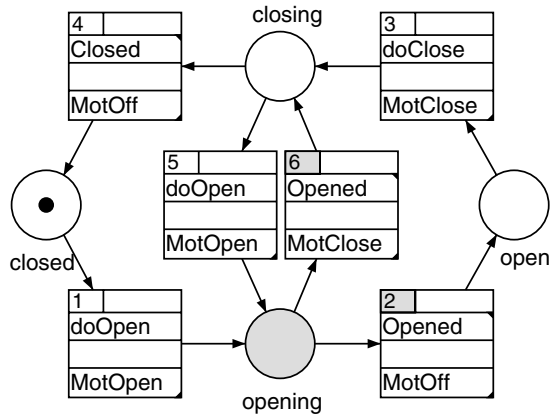
## PROCESS Operation

MODE normal

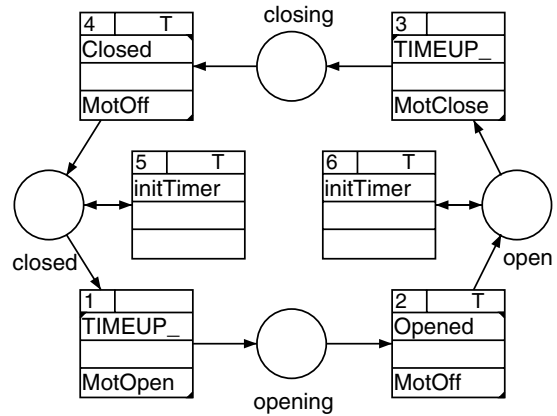


## PROCESS Door

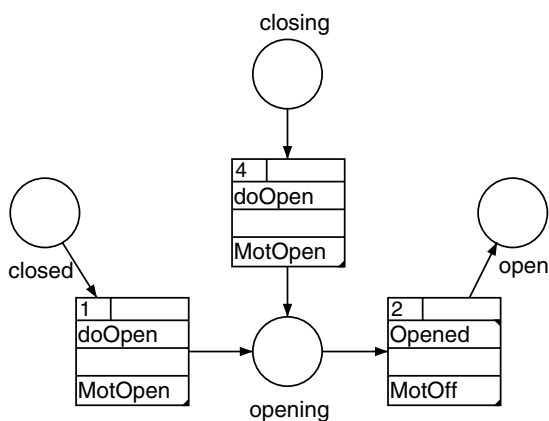
MODE userControl



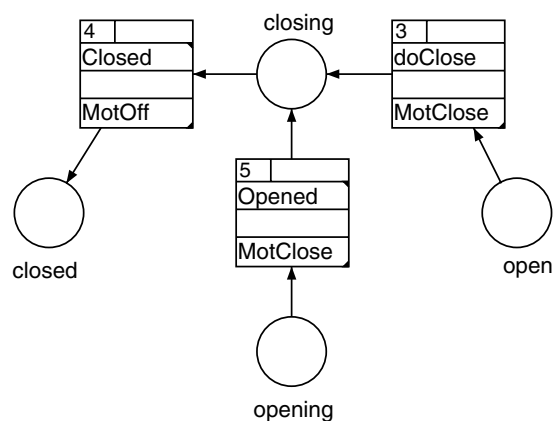
MODE scheduled



MODE holdingOpen



MODE forcedClosing



### *Mode Control Patterns*

In the following we describe some often encountered mode control patterns.

*Slave Enabled Mode Control.* If for functional reasons a mode of a process should be changed in particular states of the process only, the mode change must be enabled explicitly by means of an acknowledging pulse cast to one of the designated masters.

*Slave Controlled Mode Control.* A slave can itself initiate a mode change by sending a pulse to one of its masters to cause a particular master state transition. This pattern is used typically when an error is recognised by a slave and its master must then be triggered to induce a change to an error mode of that slave, and of other slaves as well.

*Initialisation.* Often the real initial state of an external process is not known at modelling time, although the initial state of the corresponding CIP process must be specified. The solution of this problem can be based on an initial mode of the CIP process, and on an initial state termed *init*. The first event messages from the external process is used to drive the CIP process into a state that corresponds to the real process state. When a good state is reached the process can be switched from the initial to the working mode.

### *The Difference Between Modes and Superstates*

CIP modes differ essentially from the well-known notion of super-states or serial modes of hierarchical state machine models [REF]. The modes of a CIP process are defined on the same set of states while superstates describe exclusive behaviours based on disjoint sets of states. Thus the history expressed by the current states of the lower levels cannot be retained when the pertaining superstate is changed.

A further difference to hierarchical state machines lies in the nature of the hierarchy relation. A master-slave hierarchy relation is a set of associated state machines, whereas hierarchical state machines represent compositions based on nested state sets.

### 6.7 *Modelling Sugar Supported by CIP Tool*

Useful programming language constructs that are equivalent to an expression built from elementary language constructs are often termed *syntactic sugar*. We similarly use the term *modelling sugar* for model extensions supported by CIP Tool that could also be built from elementary modelling constructs. In the following the TIMER, CHAIN and AUTO extensions are presented.

#### 6.7.1 *TIMER-EXTENSION*

A process can be extended with a TIMER. Timer delays are defined in the DELAY list of the process. A DELAY is specified by a programming language integer expression (like conditions). The resulting integer value of the evaluated expression defines the number of TICKS that must be occur until a set timer expires. The duration of a TICK is defined when the CIP model is implemented (`TICK_()` calls at the CIP machine interface).

##### **SET TIMER**

The timer can be set in any state transition of the process by marking the transition with the set timer symbol **T** and by assigning a DELAY of the delay list to the transition. When a the timer is already set, it is reset with the new delay.

##### **STOP TIMER**

The timer can be stopped (killed) in any state transition of the process by marking the transition with the stop timer symbol **S**.

##### **TIMEUP\_**

The TIMEUP\_ trigger can be assigned to any state transition of the process as input stimulus. TIMEUP\_ occurs when the set timer has expired. If an occurring TIMEUP\_ is not expected in the current process state, the input stimulus is ignored (like a not awaited impulse).

##### *Remarks:*

A TIMEUP\_ trigger is, like an input message, an external trigger that only can occur when the cluster is not performing pulse cast interaction sequence (run-to-completion semantics).

The implementation of the CIP machine determines when a pending TIMEUP\_ trigger activates the corresponding process (`TIMEUP_()` call at the CIP machine interface).

### *Example*

A lamp is turned on for 5 seconds when the lamp button is pressed. The button can also be pressed when the lamp is burning to prolong the burning time for 10 seconds. In addition, a *stop* pulse sent by another process must turn off the lamp immediately and cancel the set timer.

#### **PROCESS** TimedLamp

INCLUDE

```
#define TICKS_PER_SECOND 50 /*usually set in a header file*/
```

INPORT ButtonPort

MESSAGES On

IMPULSES stop

OUTPORT ActPort

MESSAGES Bright, Dark

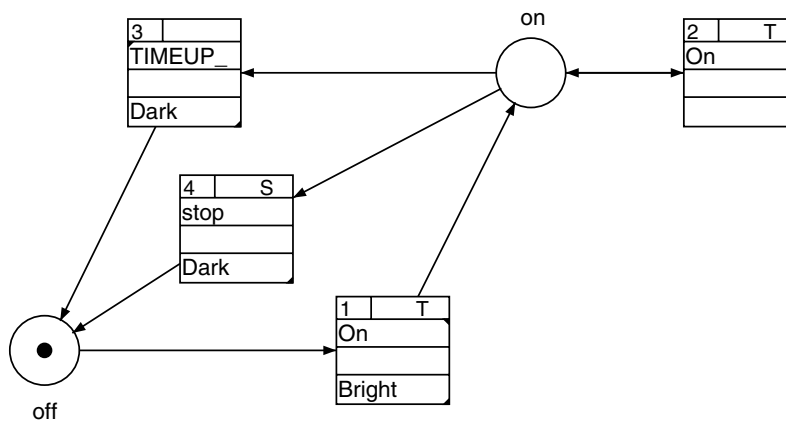
DELAYS

DELAY set5sec

```
5 * TICKS_PER_SECOND
```

DELAY set10sec

```
10 * TICKS_PER_SECOND
```



OPERATION ASSIGNEMENT

TRANSITION 1 / DELAY set5sec

TRANSITION 2 / DELAY set10sec

### 6.7.2 CHAIN -EXTENSION

A process can be extended with a trigger that activates the process itself. Setting the chain trigger is similar to setting a timer with zero delay.

#### SET CHAIN

A chain can be set in any state transition of the process by marking the transition with the set chain symbol **C**.

#### CHAIN\_

The CHAIN\_ trigger can be assigned to any state transition of the process as input stimulus. A CHAIN\_ trigger can occur as external input when the current cluster transition has been terminated (run-to-completion semantics). An unexpected CHAIN\_ trigger is ignored. When other external input messages are pending the CIP machine implementation determines the activation order. If chain triggers for several processes are pending, the process activation order is not-determined (N pending chains lead to N different cluster activation steps).

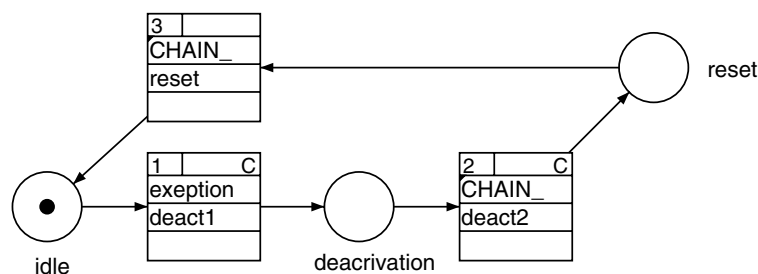
The chain extension can be used for instance to split a complex calculation of a transition into two or more pieces that are executed in subsequent state transitions (reducing the maximal cluster reaction time). The self trigger can also serve to split a complex pulse cast into simpler casts caused in subsequent transition (use two or more outpulses instead of one).

#### Example

The *ChainedExceptionManager* process must deactivate a number of devices and reset some controllers when an exception is notified.

#### PROCESS ChainedExceptionManager

MODE normal



### 6.7.3 AUTO-EXTENSION

A process can be extended with an external trigger mechanism that is active all the time.

#### **AUTO\_**

The AUTO\_ trigger can be assigned to any state transition of the process as input stimulus. AUTO\_ can occur at any time. An unexpected AUTO\_ trigger is ignored.

The AUTO\_ trigger is usually implemented with lowest priority. An AUTO\_ trigger then only occurs when no other input message, no TIMEUP\_ and no CHAIN\_ trigger are pending. Thus AUTO\_ can be used to trigger processes responsible for background work.

#### *Example*

The *AutoTriggeredWorker* process maintains two configuration tables. Each table row is updated in a separate cluster transition triggered with AUTO\_.

#### **PROCESS** AutoTriggeredWorker

MODE normal

