3 Modelling of Reactive Components with CIP

This chapter describes the graphical modelling formalism CIP which supports the domain-oriented development of reactive behavioural models. The CIP acronym means *Communicating Interacting Processes*.

A CIP model is an executable specification of concurrent reactive machines. Such models are built by creating, composing and connecting extended finite state machines. We consider CIP models that consist of singular state machine instances only (replicated instances can be specified as state machine arrays).

The compositional modelling approach is supported by CIP Tool® (http://www.ciptool.ch) 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.

Section 3.1 explains the concept of architectural composition of synchronously and asynchronously cooperating finite state machines. Section 3.2 gives an overview of the architectural structure of a CIP model. The presentation of the modelling framework starts in section 3.3 with the general CIP model structure. Section 3.4 describes the elements of an extended finite state machine, and how these elements can be related in the graphical behavioural specification. The subsequent sections define the interaction mechanisms supported by the modelling framework: asynchronous *message passing*, synchronous *pulse cast*, *state inspection* and *mode control*.

3.1 Compositional Behavioural Models

We base the domain-oriented development of embedded control functions on abstract reactive machines that model rigorously the functional system behaviour. Two main abstractions make such an approach feasible. The first of these abstractions is the abstract connection between the technical processes and the reactive machine, discussed in the previous chapter. This abstraction allows the developer to base the reactive machine on essential process phenomena of the technical processes, isolating so the functional model from properties of the peripheral devices.

The second abstraction concerns algorithms and data processing. Our abstract reactive machine is composed of a number of cooperating finite state machines. These elementary behavioural models are extended with variables, operations and conditions that are defined in the programming language of the final implementation. On the modelling level, operations and conditions represent computational primitives that are activated when the abstract reactive machine performs its state transitions.

The interaction between the cooperating state machines is modelled by means of architectural composition. This construction technique, well known from architectural description languages, uses connector artifacts to construct interaction between state machines. Connectors model interaction by means of relations between state machine interfaces. They describe interaction on the same level of abstraction as state machines describe behaviour by means of state transition relations.

This section starts with a discussion about the nature of extended state machines. The subsequent subsection compares synchronous and asynchronous cooperation of state machines. The last subsection treats the fundamental problem of constructing interaction between state machines.

3.1.1 Extended Finite State Machines: Qualitative and Quantitative Levels of Reactive Behaviour

In a classical paper, Harel and Pnueli have introduced the distinction between transformational and reactive systems to characterize the difference between common information systems and embedded real time-systems: “A transformational system accepts inputs, performs transformations on them and produces outputs. In contrast, reactive systems are repeatedly prompted by the outside world and their role is to continuously respond to external inputs.” Transformational systems may ask for additional inputs from time to time and produce outputs as they go along while reactive systems are supposed to maintain a certain ongoing relationship with its active environment.

We use extended finite state machines to specify the reactive behaviour of embedded systems. An extended state machine is a plain state machine equipped with instance variables, data fields associated to input and output symbols, and conditions and operations that are evaluated and executed in state transitions. The plain state machine describes how the state machine reacts on input stimuli while the extension enhances the state machine with computational power.

Reactive Behaviour Described by a Finite State Machine. Inputs and outputs of a reactive system are termed stimuli. Input stimuli induce instantaneous responses of the reactive system; that is, the occurrence of a stimulus and the caused reaction take place at the same point in time. In order to describe reactive behaviour, we must define the stimuli to be produced for each occurring input stimulus. In addition, we must keep information about the relevant part of the input history because reactive behaviour is in general strongly history sensitive: a reaction on an occurring input stimulus depends in general on the time order of the stimuli received in the past. The part of the stimuli history, that is relevant for reactions, can be represented by states that are updated correspondingly. Reactive behaviour can so be described by means of state transitions that are triggered by input stimuli and that produce output stimuli. This is the ubiquitous modelling notion of event driven state machines, widely used to describe the behaviour of reactive systems.

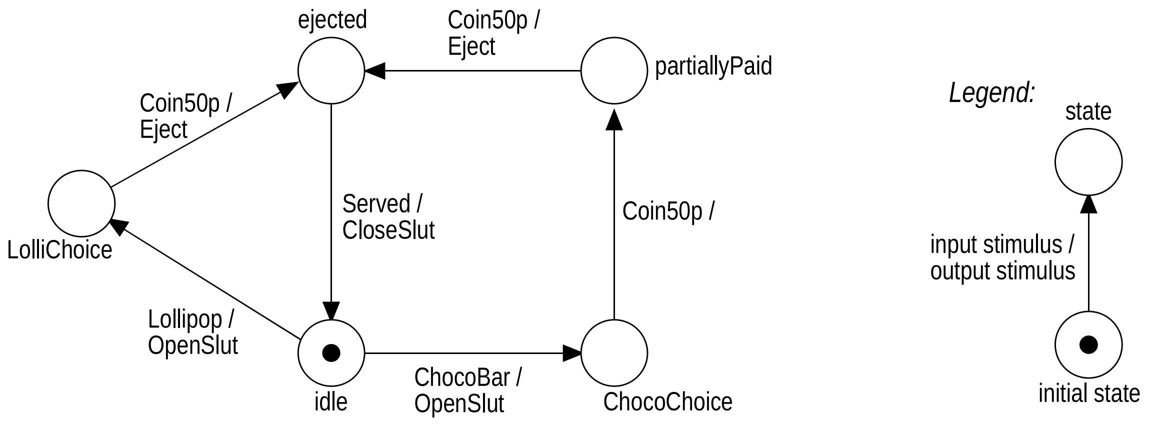
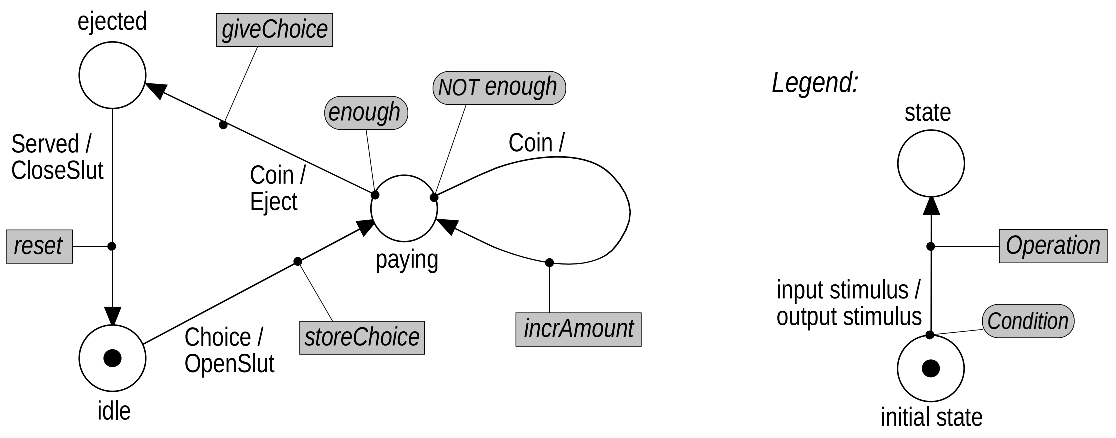


Figure 12: FSM - Finite State Machine

The finite state machine depicted in Figure 12 controls a simple selling machine for lollipops and chocobars. The reaction on inserted 50p-coins depends on the previously pressed radio button for lollipops and chocobars.

Extended Finite State Machines.

A plain state machine describes reactions by means of a transition relation that defines reaction conditions depending of input stimuli and state. In addition to this qualitative description of reactive behaviour we often need an additional quantitative description of reactions. In order to enhance a state machine with computational power, the state machine is extended by local variables, operations and conditions; stimuli are allowed to carry data. The local variables serve to store the relevant part of received input data. The stored data is often the result of complex computations performed by operations that depend of the input and the already stored information. These operations define additional behaviour on a lower level of abstraction. The production of output is correspondingly extended by operations computing the data to be carried by output stimuli.



Data Types

tProdNr	`struct{int prodNr;}`
tValue	`struct{int value;}`

Input Stimuli
Choice: tProdNr
Coin: tValue
Served

Output Stimuli
Eject: tProdNr
OpenSlut
CloseSlut

Variables

```
int price[4] = {10, 50, 100, 200};
int choice;
int amount;
```

Operations

storeChoice	`choice = IN.prodNr;`
incrAmount	`amount = amount + IN.value;`
giveChoice	`OUT.prodNr = choice;`
reset	`choice = 0; amount = 0;`

Conditions

enough	`amount+IN.value >= price[choice]`

Figure 13: EFSM - Extended Finite State Machine

The extended finite state machine of Figure 13 controls a selling machine that offers four different products. The prices of the individual products are stored in the statically initialized array variable *price*. The integer variables *productChoice* and *amount* are used to store the selected product and the amount of money inserted by coins. The input stimuli *Choice* and *Coin*, and the output stimulus *Eject* carry the data declared by the associated data type. The data are handled by the various operations associated to state transitions. The condition *enough* is used to resolve the non-deterministic behaviour in the state *paying*.

A state machine is deterministic if in every state at most one state transition can be triggered by a particular input stimulus. If a state machine is non-deterministic, the non-determinism can be resolved by exclusive guard conditions associated to state transitions. Such conditions can in general depend on the input data and local variables of the state machine. Although the basic plain state machine is non-deterministic, the extended state machine can resolve the ambiguity and represent a deterministic behavioural component. For example, the state machine of the selling machine in Figure 13 is non-deterministic in the state *paying* because two different state transition can be triggered by the *Coin* input stimulus. The condition *enough* used to resolve the non-determinism depends on the coin value, on the stored amount, and of the price of the selected product.

Domain-oriented description of reactive behaviour, as widely recognised and applied, is therefore naturally based on extended finite state machines. An extended state machine can be viewed as plain state machine with plugged computational primitives. The basic plain state machine acts as operational framework that triggers locally integrated computations during state transitions.

Plain state machines are abstract machines suited to model reactive behaviour qualitatively. The modelling level is determined by the chosen set of input and output stimuli. Quantities handled during a reaction are described on a lower level of abstraction by means of algorithmic operations. As the purpose of programming languages is describing computations, programming languages suite well for the quantitative description of reactive behaviour. Viewed from the modelling level of the plain state machine, operations are computational primitives that represent uninterpreted values. Their meaning is defined by the semantics of the used programming language and their description cannot be interpreted at the modelling level.

The qualitative and quantitative description of reactive behaviour supported by extended state machines have different characteristics. The qualitative description supported by the plain state machine model does in general not show much regularity. It is typically the result of distinguishing a bounded number particular cases. The quantitative description is usually much more regular. Due to its algorithmic nature it can be based on computation rules, calculations and data processing functions.

3.1.2 Asynchronous and Synchronous Cooperation of State Machines

If we try to model the reactive behaviour of an embedded system by means of a singular state machine, we will soon recognise that the model will become extremely complex and incomprehensible. A main reason is that the system environment consists of several active physical processes. Due to the concurrent behaviour of these processes, the state machine must capture a corresponding number of parallel event histories. The result is inevitably a state machine with a huge number of states and transitions; this effect is usually called *state space explosion*.

Each state of such a state machine typically represents the combined information about the history of the individual processes. The well known recipe to disentangle independent time orders of events is to build models that consist of several cooperating state machines. Coordination of individual state machines can be specified by creating interaction between these individual behavioural components.

Cooperation of state machines can take place either asynchronously or asynchronously. Asynchronous cooperation, supported by parallel modelling languages like SDL or ROOM, is necessary to support distributed implementations while synchronous cooperation, well known from many real-time description techniques such as Statecharts and Esterel, is needed to model deterministic propagation of internal interactions.

Asynchronous Cooperation.

Asynchronously cooperating state machines represent concurrent behavioural components with no a priory notion of global time and global state. Interaction chains take place concurrently. However, if a component must participate at the same time in more than one interaction, a serious synchronisation problem occurs. Asynchronous interaction mechanisms are either blocking or non-blocking. Blocking mechanism block the initiator of an interaction until the activated component participates in the interaction. Well known examples are the Rendez-Vous mechanism of ADA and the mechanism of remote method invocation of active objects. Non-blocking interaction mechanism, in contrast, are based on temporally separated actions, such as write into a buffer and read from the buffer. Examples are the SDL channels or the well known FIFO message queues of real-time operating systems.

The advantage of asynchronous cooperation is the important number of possible implementation configurations, supported by tasks running on a singular or several processors. The main problem of asynchronous cooperation in reactive systems is that different interaction chains can interleave; that is, several chain reactions can be concurrently active. Models based on asynchronous cooperation are therefore basically non-deterministic, even when the cooperating state machines are implemented on a singular processor. In addition to the problem of modelling unique system reactions, non-deterministic execution of interaction sequences makes it in general very difficult to determine worst case execution times for chain reactions caused by external events. Handling these difficulties is the price to be paid for the gained implementation freedom.

Synchronous Cooperation.

Synchronously cooperating state machines represent synchronous components of a composed state machine whose states are given by the state combinations of these components; i. e. to give the state of the composed state machine one must give the state of every component. A mathematician would say, the state space of the composition is the Cartesian product of the component state spaces. High priority internal interaction mechanisms, usually based on stimuli broadcast, allow specifying system reactions as internal chain reactions that are not interruptible by external input stimuli. An active internal interaction propagation is always completed before the next external input stimulus is accepted. This kind of behavioural semantics is called *run-to-completion semantics*. The state machine composition behaves itself as a state machine; it processes external stimuli one by one. Each chain reaction, triggered by an external stimulus, can due to the run-to-completion semantics be viewed as a singular state transition of the composed state machine. Synchronously cooperating state machines are implemented usually in a single sequential task.

The advantage of synchronous cooperation is that it is possible to define deterministic interaction models; if all components of such a model are deterministic then the resulting composition can be forced to be deterministic too. There is no interleaving of internal interaction chains, and it is even possible to deduce worst-case execution sequences from the model. This is exactly what we need when a local group of physical processes have to be controlled by an embedded system. A disadvantage of synchronous cooperation is that it is difficult to implement such a model on several processors.

Combination of Asynchronous and Synchronous Cooperation

CIP unifies asynchronous and synchronous cooperation of state machines within the same model. The reactive machine defined by a CIP model is based on a number of asynchronously cooperating functional blocks termed clusters. Each of these clusters consists of a number of synchronously cooperating extended finite state machines. Clusters are used to model deterministic software components that control process groups with a strong functional coherence. In addition, the system model can be easily implemented on distributed target systems as long as no singular cluster is implemented on more than one processor.

3.1.3 Architecture-Based Modelling of Interaction

We claim to construct the abstract reactive machine by means of architectural composition, that is: elementary state machines are interconnected by specific connectors which mediate interaction and communication between these components. The flexible compositional approach supports strongly the development of intelligible and maintainable interaction structures of embedded systems.

Modelling of Interaction

Machines are systems that consist of interacting parts. Interaction is always based on particular type of phenomena that take place between the interacting parts. Mechanical, electrical and hydraulic machines are examples of physical machines that are based each on a particular interaction types. The type of the used interaction determines the kind of machines that can be constructed. Thus when a particular machine has to be developed, a main engineering choice concerns the type of interaction that is to be used. This should not be different when we construct abstract machines.

The elementary components of our abstract machine are extended finite state machines. Each of these components represents an elementary state machine that is extended with computational primitives. By constructing interactions between the basic plain state machines we obtain the desired abstract reactive machine that models the global collective behaviour of the embedded system. In order to get a rigorously described abstract machine, interactions must be defined on the same level of abstraction as the plain state machines; for example by relating the output stimuli of one state machine to the input stimuli of an other state machine. In order to support an interaction that is related also to the computational extension of individual state machines, the modelled interaction is extended by mechanisms that allow the transport of data-items described in a programming language. The result are abstract reactive machines enhanced with computation power; that is, for the composed system we have a similar distinction of quantitative and qualitative reactive behaviour as for the elementary extend state machines.

By defining state machines as components we can build compositional system models. Components are system parts that are not changed when interaction is defined; interaction takes place between the components. Every component, including its interface, must be inert with respect to composition. A change at the interface of a component, even a simple renaming of an input symbol, must not have any side-effect on the components connected by interaction.

In order to be able to define interactions between components we need a further category of modelling artifacts called connectors. The resulting approach of building architectural compositions of interacting components is presented briefly in the next subsection. The compositional construction technique allows the developer to construct reactive machines similar to physical machines, i. e. by interconnecting submachines by means of specific connectors that mediate interaction.

Architecture-based Models of Interaction

The rigorous description of software architecture by means of *Architecture Description Languages* (*ADL*) is a new emerging discipline of software engineering. Rigorous architectural system description separates the description of local behaviour and interaction. CIP adopts the concept of ADL's for the description of cooperating state machines.

Architectural system description is well known from the widely used box-and-line diagrams, for instance to describe analysis models. The boxes, or circles, of such diagrams represent computational entities with behaviour and state whereas the lines represent interaction connections. However, the problem with conventional analysis models is that neither the behaviour of components nor the interaction mechanisms associated with connectors are defined rigorously.

ADL's base the construction of software systems on rigorous *architectural composition*: that is, the global system behaviour is defined by composing behavioural components by means of interaction connectors, as shown in Figure 14. Components represent behavioural artifacts that are inert with respect to composition. Connectors mediate interaction between components and define so how the behaviour of the singular components combine to the global behaviour of the composed system.

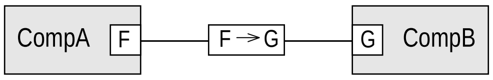


Figure 14: Interaction connector connecting two components

The basic elements of architectural description are components, connectors and configurations:

Components
Components represent the computational entities of a system. They are the locus of behaviour and state. The interface of a component is defined by a set of ports which define the component's point of interactions with its environment.

Connectors
Connectors represent first class artifacts defining the interaction between components. They are the locus of relations between component interfaces. Connectors ends have associated roles that define how an attached component can participate in the specified interaction.

Configuration
A configuration is a collection of components which interact by means of connectors. Connector ends are attached to ports of components.

We use architectural composition to construct abstract reactive machine that define the functional behaviour of embedded systems. Our components are extended state machines, and a number of connector types are used to construct interaction between state machines. The principle of separate specification of local behaviour and interaction leads to a compositional system description. The behaviour of a composition can always be deduced from the behaviour of the components and their interconnections.

The architectural approach supports perfectly the construction process stipulated by the correspondence concept which bases the development of reactive machines on an embodied context model. The collective behaviour of the reactive machine is obtained by interconnecting independent context agents with components responsible for coordination.

3.2 Architecture and Structure of CIP Models

This introductory section gives a first view on the architectural modelling approach of the CIP method. The notions used in the overview are developed and explained in the following sections of this chapter.

3.2.1 Architecture of CIP Models

CIP combines asynchronous and synchronous composition of state machines within the same model. A CIP model consists of a set of asynchronously composed *clusters*, each consisting of a number of synchronously composed extended finite state machines termed *processes*. Asynchronous composition is necessary to support the distributed implementation of generated software components. Synchronous composition, on the other hand, is needed to model deterministic reactive components with bounded response time.

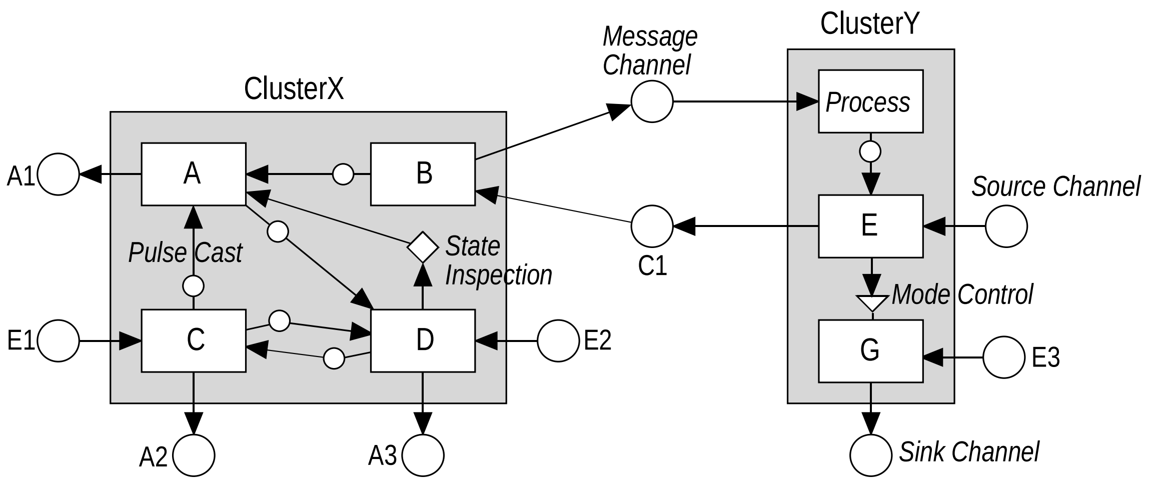


Figure 15: Clusters, processes and interaction connectors of a CIP model

Process cooperation is specified by means of interaction. Figure 15 shows the architecture of a CIP model example. The processes of the two clusters are connected by various interaction connectors. Message channels connect processes that interact by asynchronous message passing. In addition, source and sink channels model the abstract connection to the physical processes of the system environment (technical processes domain).

Within a cluster, processes stimulate each other by means of sequential pulse cast interaction. State inspection and mode control represent additional interaction mechanisms that allow to define state dependencies between the state machines of the same cluster. Processes and channels can be instantiated within process and channel arrays. The various interaction types support selection mechanisms that allow addressing particular instances of connected process arrays.

3.2.2 CIP Model Structure

A CIP model consists of several concurrent CLUSTERS, viewed in Figure 16 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.

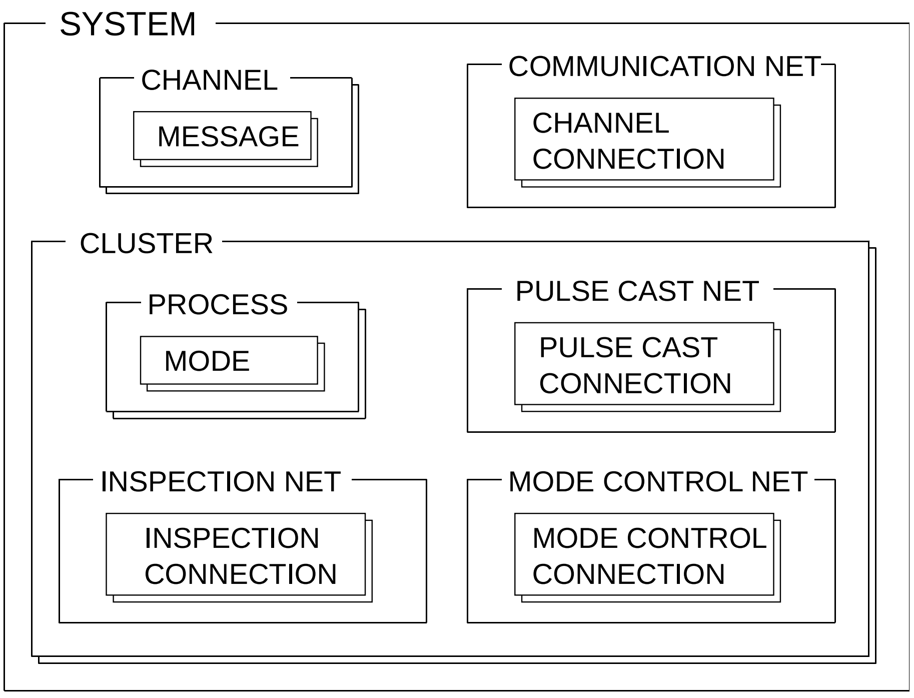


Figure 16: Components and interaction connections of a CIP model

3.3 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.

3.3.1 Modelling Elements of a Process

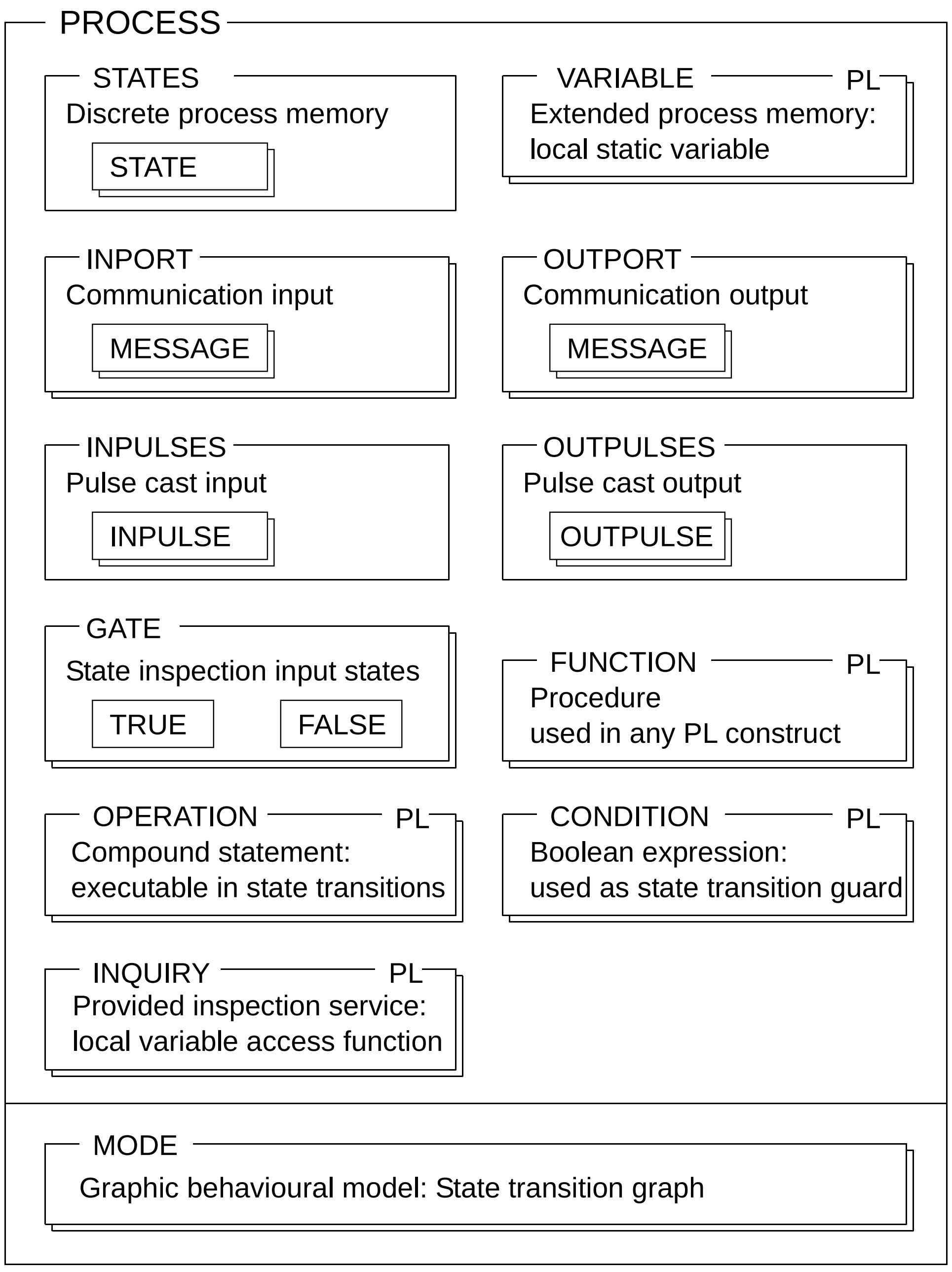


Figure 17: Process elements

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 17 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 – OUTPULSES, INPULSES

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 outpulse appears as received inpulse 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 – INQUIRIES

An inquiry of a process is a procedure which can be called by the processes of the same cluster. For the callers, a corresponding *state vector inspection connection* (inspection net) has to be specified. An inquiry is defined as a procedure (PL) which delivers information depending on the current state vector of the inspected process.

Process Behaviour – MODES

The process behaviour is defined by one ore 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 outport. 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.

3.3.2 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 17 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 18 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

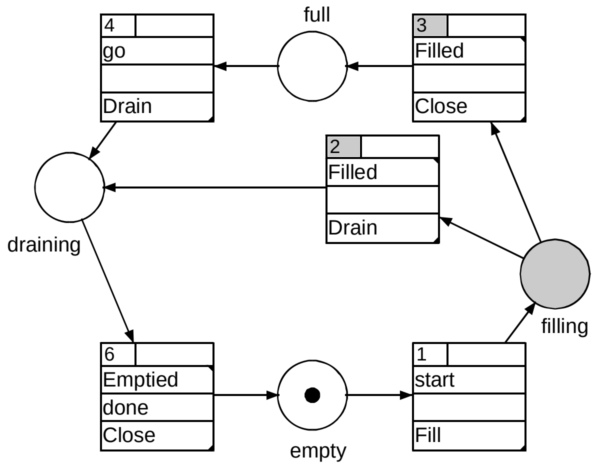
INPULSES
go, start

OUTPULSES
done

OUTPORT ValveActions
MESSAGES Close, Fill, Drain

GATES
grantDraining

MODE normal



SWITCH

STATE filling MESSAGE Filled
TRANSITION 2 / grantDraining
TRANSITION 3 / ELSE_

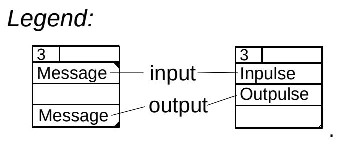


Figure 18: CIP specification of the SimpleVesselCtrl process

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* inpulse 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.

3.3.3 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 19. 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* inpulse 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).

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.

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.

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 ExtendedVesselCtr

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

STATES
draining, filling, full. empty

VARIABLES
order: tInteger, counter: tInteger

INPORT LevelEvents
    MESSAGES Emptied, Filled

INPULSES
go, start: tInteger

OUTPULSES
done

OUTPORT ValveActions
    MESSAGES Close, Fill, Drain: tFloat

GATES
grantDraining

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

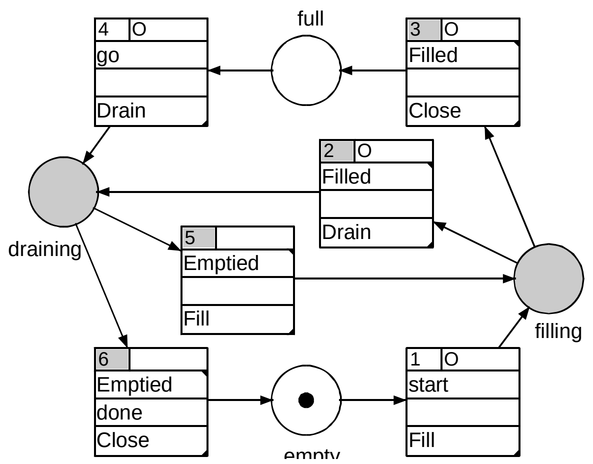
OPERATION initCounter
```
{SELF.counter = 0;}
```

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

OUTPORT OPERATION giveDrainFlow
```
{OUTMSG = Config.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 ASSIGNEMENT

TRANSITION 1 / OPERATIONS storeOrder, initCounter

TRANSITION 2 / OPERATIONS incrementCounter
    OUTPORT ValveActions OPERATIONS giveDrainFlow

TRANSITION 3 / OPERATIONS incrementCounter

TRANSITION 4 / OUTPORT ValveActions OPERATIONS giveDrainFlow

Figure 19: 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* inpulse, 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 ‘im Anhang’.

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 (see section 3.8).

3.4 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.

3.4.1 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 16 of section 3.2.2. 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 20 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.

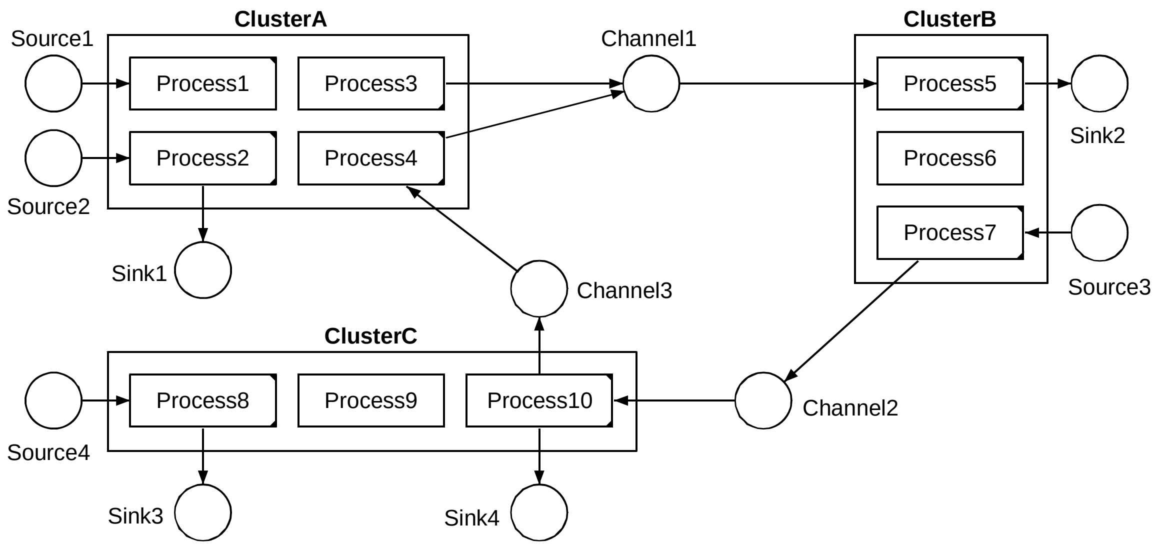
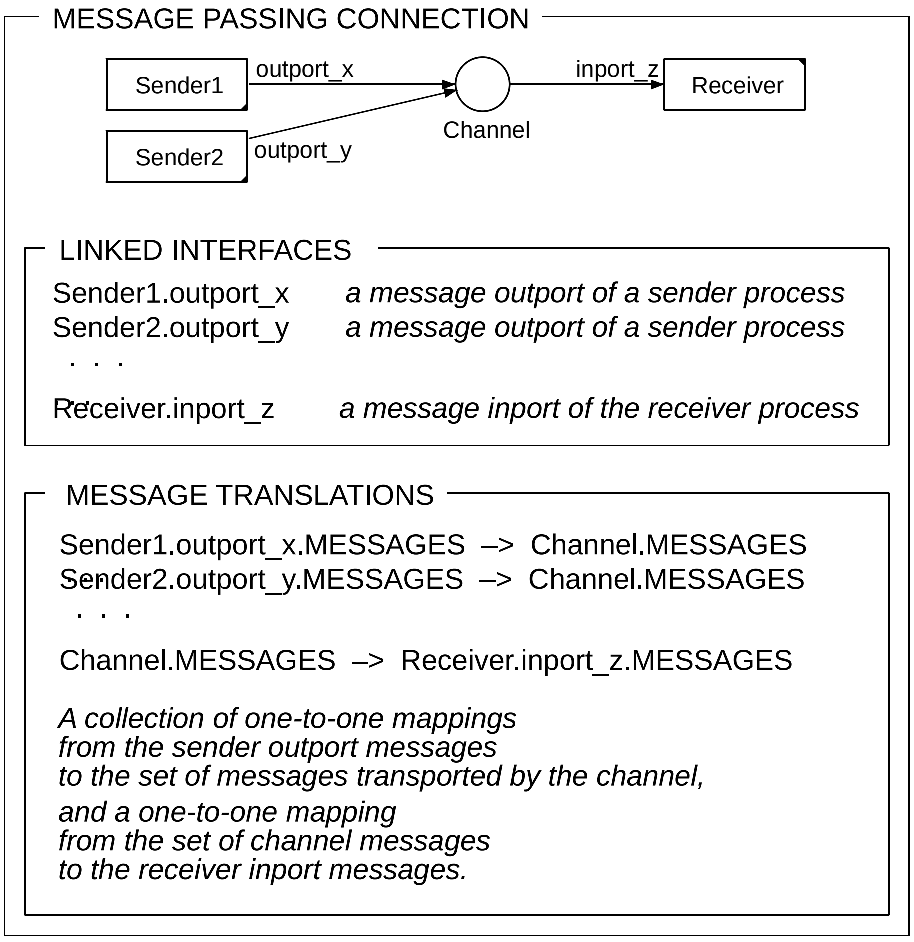


Figure 20: COMMUNICATION NET – Processes linked with channels

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.

The frame of Figure 21 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 outports 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 outport messages as a unique message to the receiver inport, outport, channel and inport message must be related one-to-one. Thus the number of outport messages, the number of channel messages and the number of inport messages of a connection must be equal.



Sender1.outport_x	*a message outport of a sender process*
Sender2.outport_y	*a message outport of a sender process*
. . .	
Receiver.inport_z	*a message inport of the receiver process*

Sender1.outport_x.MESSAGES –> Channel.MESSAGES
Sender2.outport_y.MESSAGES –> Channel.MESSAGES
. . .
Channel.MESSAGES –> Receiver.inport_z.MESSAGES

A collection of one-to-one mappings from the sender outport messages to the set of messages transported by the channel,

and a one-to-one mapping from the set of channel messages to the receiver inport messages.

Figure 21: 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 support 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 outport, channel and input messages are related: messages associated by message translations must have the same data type.

3.4.2 The Message Passing Interaction Mechanism

A channel is an interaction connector that accepts outport messages from one ore 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 outport 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 activation 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 4).

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 chapter 4.

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 connections between the generated CIP Units and 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 (*CIP Shell*) of each CIP Unit.

The modelled connections to the system environment (source and sink channels) are to be implemented by interface devices attached to the external processes, and by active device interface modules which connect interface devices and CIP Shell of a CIP Unit. 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.

3.4.3 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.

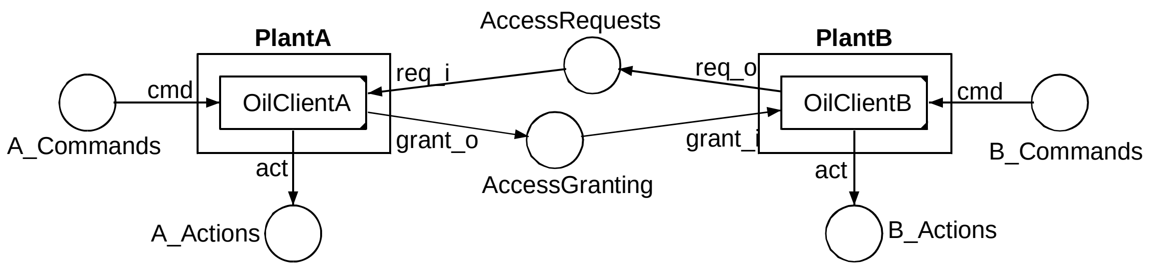
The PipelineAccessControl Problem – PAC

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 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 PiplineAccessControl_AsymmetricSolution

COMMUNICATION NET

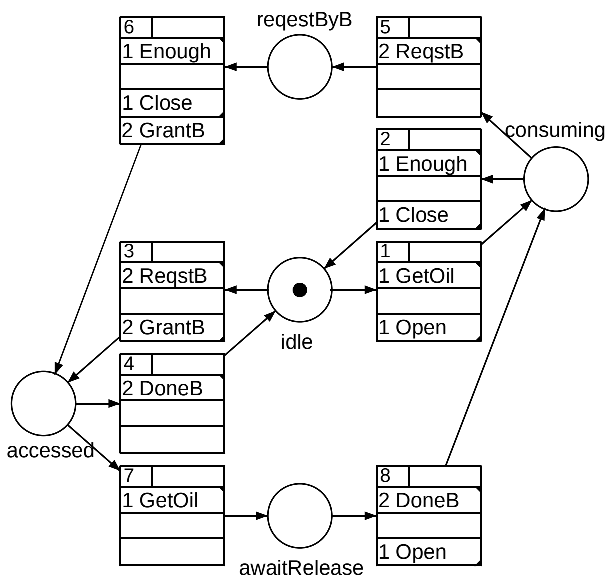


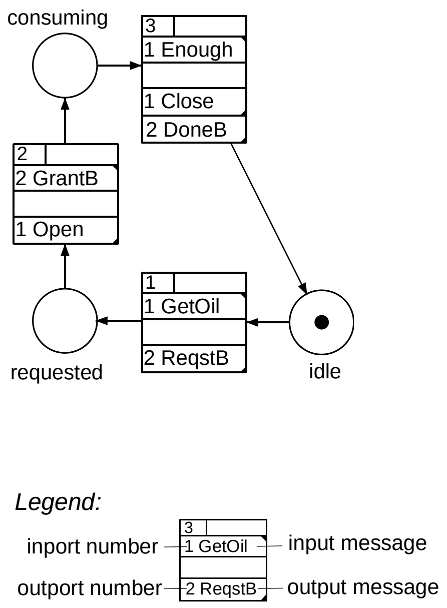
CLUSTER PlantA

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





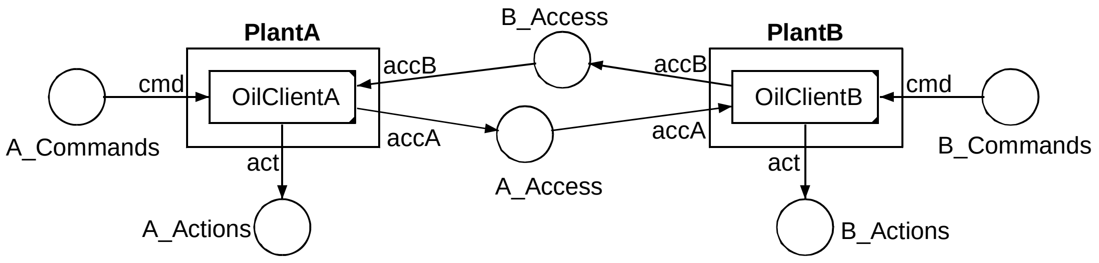
The *OilClientA* process masters the exclusive pipeline access. When the *OilClientB* process is commanded to deliver oil (*GetOil*), the process request pipeline access permission by sending the *ReqstB* message to the *OilClientA* process. When the requested permission is confirmed by the *GrantB* 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 PiplineAccessControl_IncorrectSolution

COMMUNICATION NET



CLUSTER PlantA

PROCESS OilClientA

INPORT 1 cmd
 MESSAGES Enough, GetOil

INPORT 2 accB
 MESSAGES DoneB, UserB

OUTPORT 1 act
 MESSAGES Close, Open

OUTPORT 2 accA
 MESSAGES DoneA, UserA

MODE normal

CLUSTER PlantB

PROCESS OilClientB

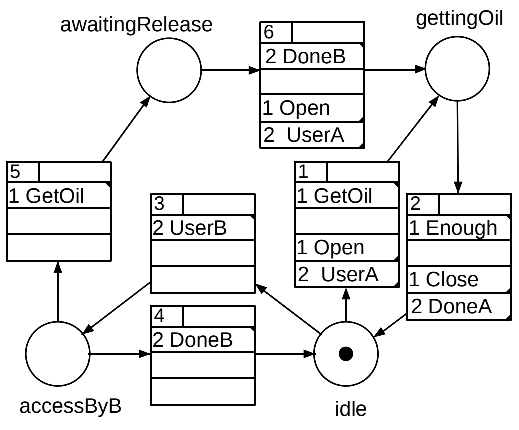
INPORT 1 cmd
 MESSAGES Enough, GetOil

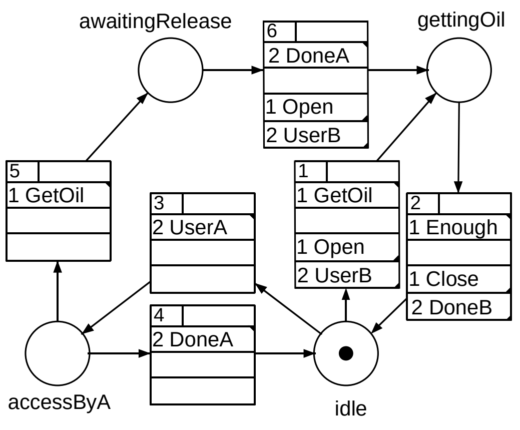
INPORT 2 accA
 MESSAGES DoneA, UserA

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.

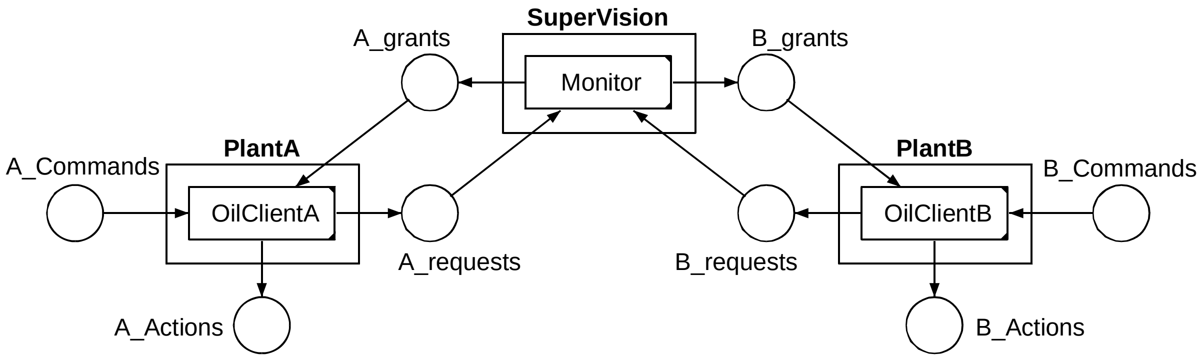
C. The Generic Monitor Solution

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.

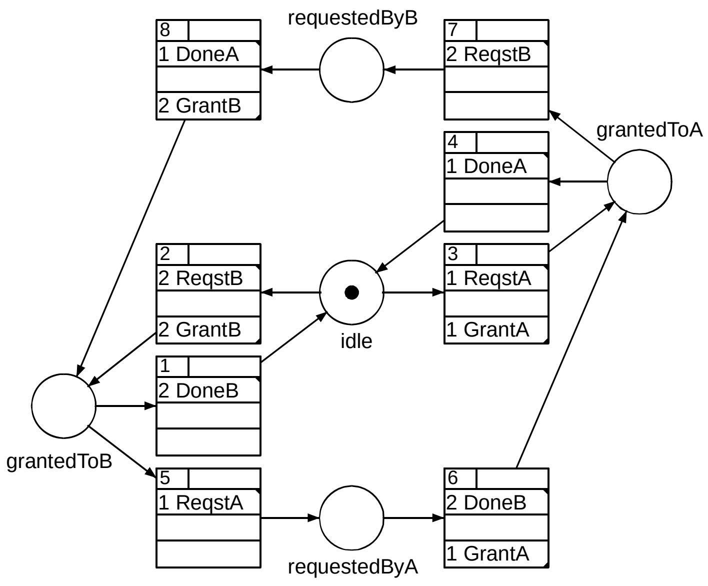
SYSTEM PiplineAccessControl_MonitorSolution

COMMUNICATION NET



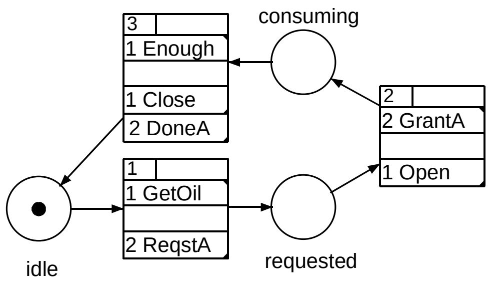
CLUSTER SuperVision

PROCESS Monitor



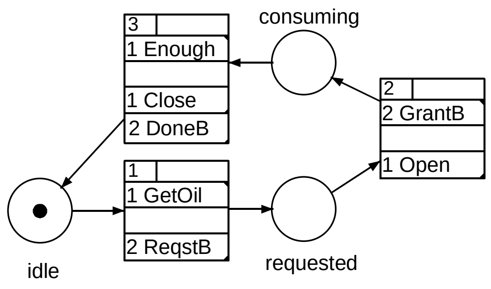
CLUSTER PlantA

PROCESS ClientA



CLUSTER PlantB

PROCESS ClientB



3.5 Pulse Cast Interaction

Pulse cast interaction is used to cause stimuli transmission between the processes of a cluster. Pulse casting is always initiated by the 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 by triggered externally by an input message (run-to-completion semantics).

3.5.1 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 22). 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 OUTPULSES and the set of INPULSES of a process represent the output and the input interface dedicated to pulse cast interaction.

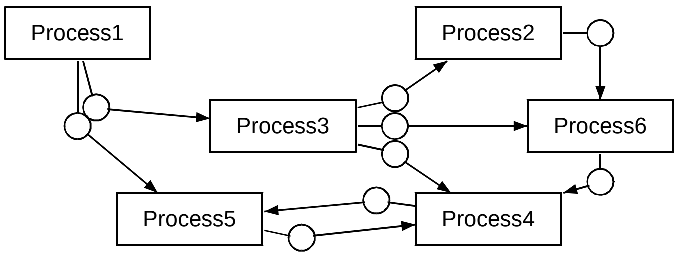
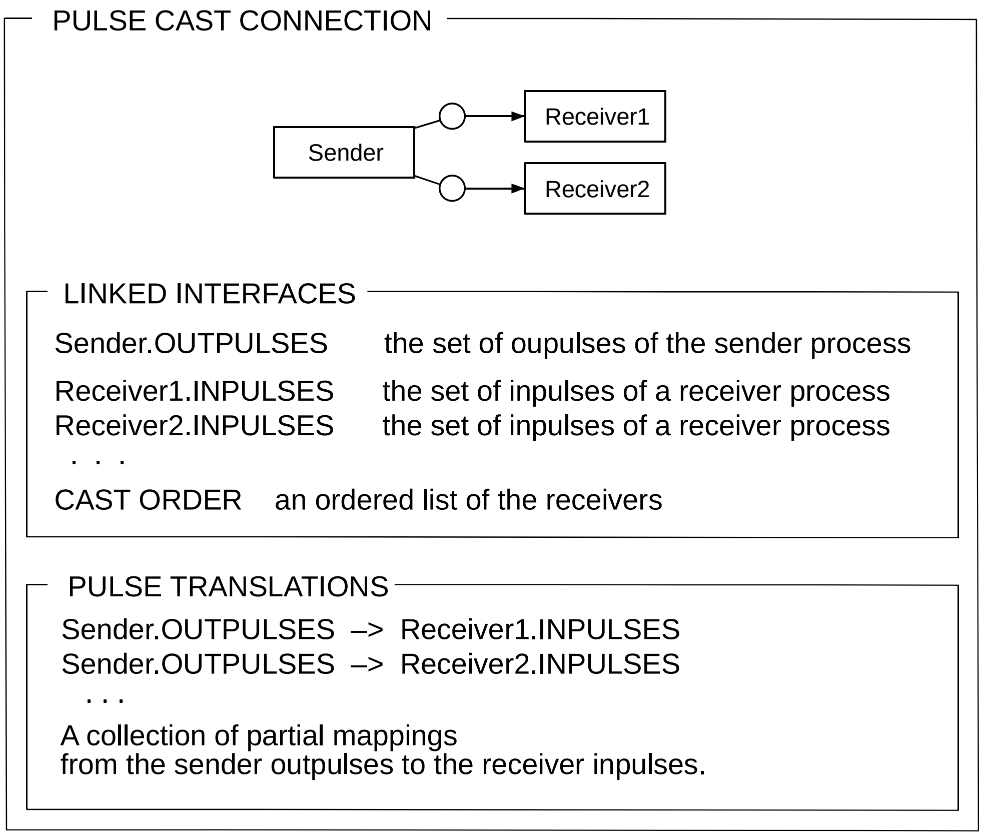


Figure 22: PULSE CAST NET – Processes linked with pulse cast connectors

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 outpulses of a sender must by mapped to inpulses 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 23 shows all elements of a pulse cast connection. The LINKED INTERFACES of a connection consist of the set of outpulses of the sender and set of inpulses of the receivers. The CAST ORDER is defined as ordered receiver list.



LINKED INTERFACES

Sender.OUTPULSES	the set of oupulses of the sender process
Receiver1.INPULSES	the set of inpulses of a receiver process
Receiver2.INPULSES	the set of inpulses of a receiver process
. . .	
CAST ORDER	an ordered list of the receivers

PULSE TRANSLATIONS

Sender.OUTPULSES –> Receiver1.INPULSES
Sender.OUTPULSES –> Receiver2.INPULSES
. . .

A collection of partial mappings
from the sender outpulses to the receiver inpulses.

Figure 23: 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 multicast 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 (multicast).

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).

3.5.2 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 s*equential* 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 3.2, 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 inpulses 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.

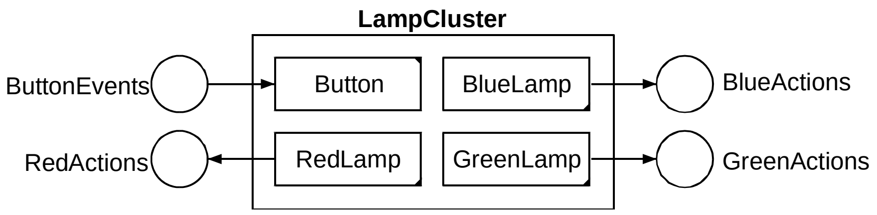
Two questions arise immediately. Is the sequence of process steps within a cluster step uniquely determined, and how is the termination of such a sequence of process steps ensured? The answer to these questions is given below in the subsection about execution rules of pulse cast chains. The deterministic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

3.5.3 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.

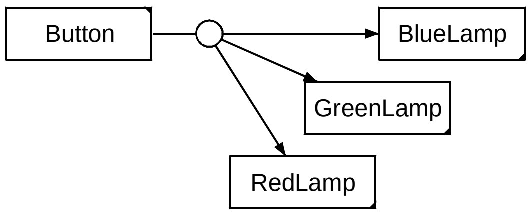
SYSTEM SimplePulseCast

COMMUNICATION NET



CLUSTER LampCluster

PULSE CAST NET



PULSE TRANSLATIONS

SENDER Button

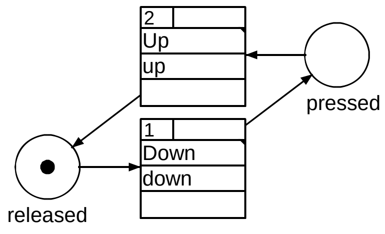
```
down -> BlueLamp.on
     -> GreenLamp.change,
     -> RedLamp.change,

up   -> BlueLamp.off,
     -> GreenLamp.change
```

PROCESS Button

INPORT EventPort
  MESSAGES Down, Up

OUTPULSES down, up

MODE normal

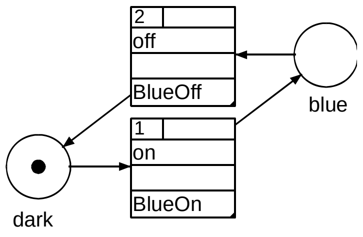


PROCESS BlueLamp

INPULSES off, on

OUTPORT ActionPort
  MESSAGES BlueOff, BlueOn

MODE normal

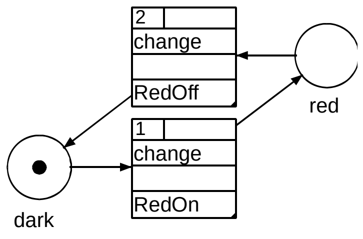


PROCESS RedLamp

INPULSES change

OUTPORT ActionPort
  MESSAGES RedOff, RedOn

MODE normal

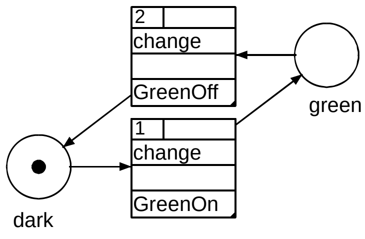


PROCESS GreenLamp

INPULSES change

OUTPORT ActionPort
  MESSAGES GreenOff, GreenOn

MODE normal



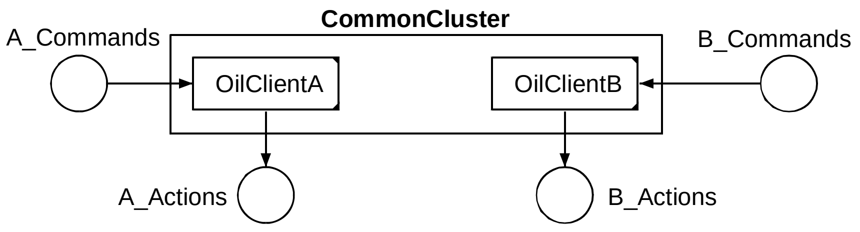
Each of the three pulse translations from the *Button* to the lamp processes is of a different type: *one-to-one*, *all-to-one* and *partial one-to-one*. Thus the blue and the red lamp are turned off and on when the button is pressed and released whereas the red lamp changes its state when the button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

3.5.4 The Pulse Cast Solution of the PipelineAccessControl Problem

The incorrect solution *B* of the *PipelineAccessControl* problem (see 3.4.3) 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.

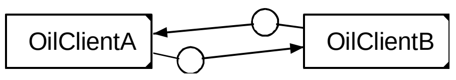
SYSTEM PiplineAccessControl_PulseCastSolution

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

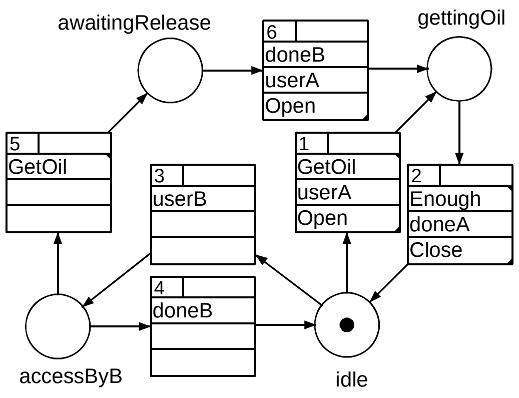
INPORT cmd
MESSAGES Enough, GetOil

INPULSES doneB, userB

OUTPULSES doneA, userA

OUTPORT act
MESSAGES Close, Open

MODE normal



PROCESS OilClientB

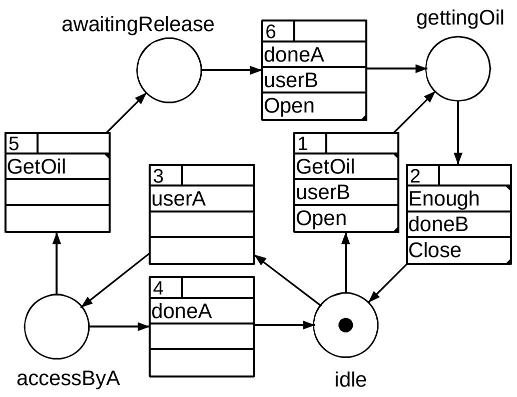
INPORT cmd
MESSAGES Enough, GetOil

INPULSES doneA, userA

OUTPULSES doneB, userB

OUTPORT act
MESSAGES Close, Open

MODE normal



3.5.5 Execution Rules for Pulse Cast Chains

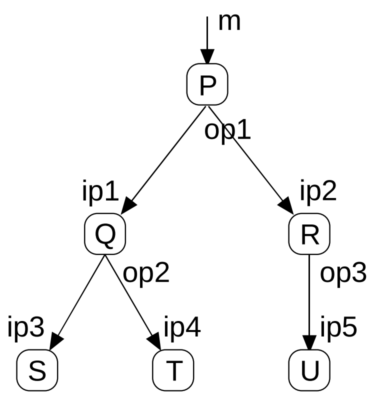
As already mentioned above, the process steps of a cluster step must be executed in a deterministic way, and the maximal number of potential process steps within a cluster step must be bounded. Specifying deterministic execution of cooperating behavioural entities is a fundamental difficulty in the development 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 *PipelineAccess-Control* 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 24. Each graphic node of the tree structure represents a potential process step to be executed within the caused 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 m1 triggers process P and the inpulses ip1, ip2, ... activate the other processes. The symbols op1, op2, ... 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.



PULSE TRANSLATIONS
P.op1 -> Q.ip1, R.ip2
Q.op2 -> S.ip3, T.ip4
R.op3 -> U.ip5

Figure 24: 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.

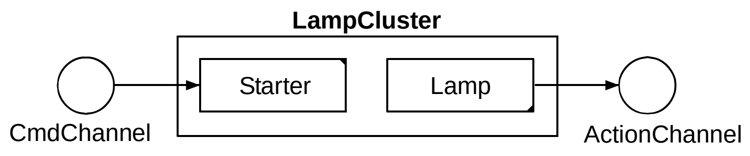
The above introduced run-to-completion semantics of cluster steps appears now as a consequence of the stronger run-to-completion semantics of chain reactions: a cluster step is the chain reaction caused by the receiving of an external message.

Bounded Cluster Steps – Non-Recursive Pulse Cast Interaction

The notion of cluster step run-to-completion semantics implicitly supposes that 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 25 shows a simple model with a *recursive* pulse cast interaction.

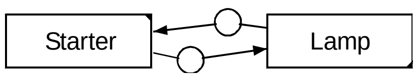
SYSTEM RecursivePulseCast

COMMUNICATION NET



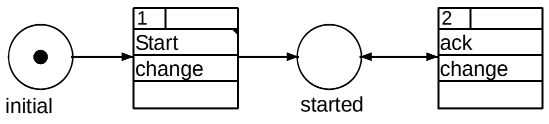
CLUSTER LampCluster

PULSE CAST NET



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

PROCESS Starter
INPORT Commands
MESSAGES Start
INPULSES ack
OUTPULSES change
MODE normal



PROCESS Lamp
INPULSES change
OUTPULSES ack
OUTPORT Actions
MESSAGES Bright, Dark
MODE normal

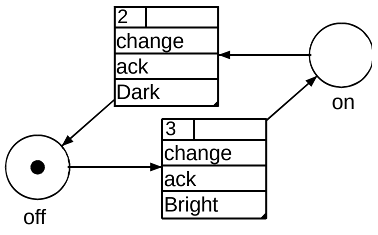


Figure 25: Example of recursive pulse-cast interaction

When the *Start* message occurs, the *Starter* process triggers with the *change* outpulse a first *change* reaction of the *Lamp* process. The activated *Lamp* process acknowledges the caused *change* inpulse with an *ack* outpulse. Because the subsequent *Starter* reaction on the received *ack* inpulse causes by the emitted *change* outpulse again a *change* inpulse, an infinite repetition of *change/ack -> ack/change* cycles occurs. Such a behaviour is termed a *live-lock.*

The repeated *change/ack -> ack/change* cycle of our example represents a recursive pulse cast path because the reaction of the *Lamp* process on the *change* inpulse causes a pulse propagation that activates the *Lamp* process again with the *change* inpulse. 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 inpulse.

If pulse cast interaction is non-recursive, all cluster steps are bounded because the number of process inpulses 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 inpulse; 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 inpulse, 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 outpulses cause transmitted inpulses, and how activated processes can react on inpulses by generating outpulses. 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 acylicity is not a severe restriction in real systems.

Pulse Cast Tree Viewer of CIP Tool. The restriction to acyclic pulse propagation paths allowed to automate pulse cast interaction analysis. A graphical *Pulse Cast Tree Viewer* of CIP Tool shows during modelling all potential pulse cast trees of a Cluster.

Remark. The mathematical description of *pulse propagation paths* and the foundation of the automated interaction analysis of CIP Tool is described in a paper available at http://www.ciptool.ch > publications > *Interaction Analysis of Reactive Models*. 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.

3.5.6 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 be turned on by the operator.

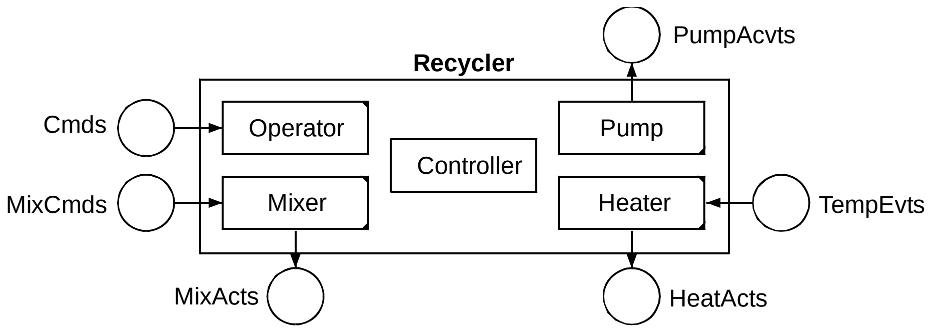
The *ChainedPulseCast* model consists of the single *Recycler* cluster only. The *Operator, Mixer*, *Heater* and *Pump* process represent context agents that are connected to the technical processes of the embedded system. Internal interaction between context agents always takes place via the *Controller*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

The purpose of the example is to show that a process can be activated several times within one cluster reaction. The pulse cast interaction in this simple example will therefore get more complex than necessary. The reason is, that all relevant state changes are captured by the *Controller* Process, be means of pulse cast interaction.

However, the state inspection interaction introduced in the next section will provide 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 always leads to shorter pulse cast chains.

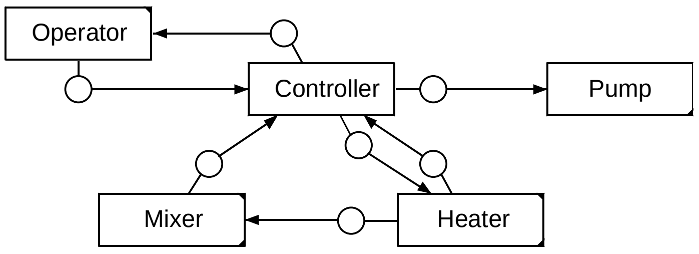
SYSTEM ChainedPulseCast

COMMUNICATION NET



CLUSTER Recycler

PULSE CAST NET



A *many-receiver* pulse cast connection is represented by a 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

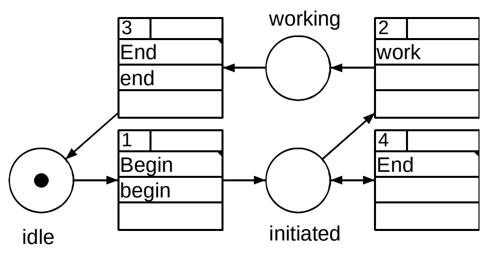
SENDER Mixer
    mixAck -> Controller.mixAck

SENDER Operator

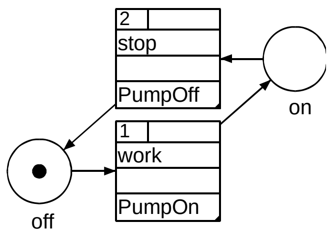
begin	-> Controller.begin
end	-> Controller.end

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

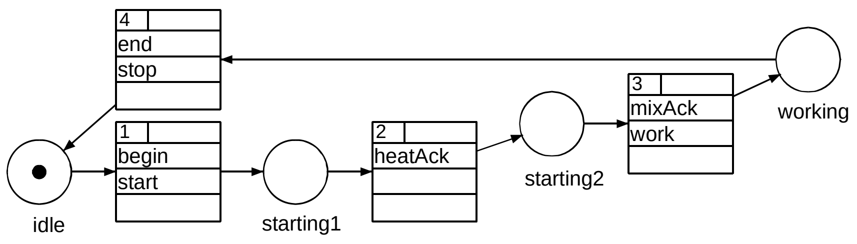
PROCESS Operator
INPORT OpCmds
    MESSAGES Begin, End
INPULSES work
OUTPULSES begin, end
MODE normal



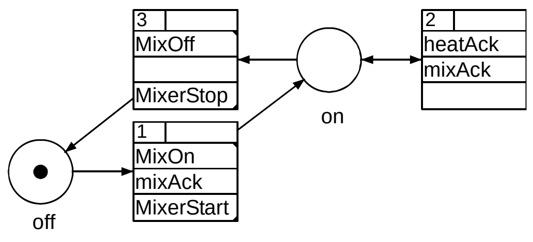
PROCESS Pump
INPULSES stop, work
OUTPORT PumpActions
    MESSAGES PumpOff, PumpOn
MODE normal



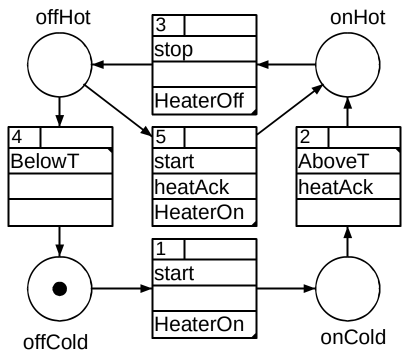
PROCESS Controller
INPULSES begin, end, heatAck, mixAck
OUTPULSES start, stop, work
MODE normal



PROCESS Mixer
INPORT MixCmds
    MESSAGES MixOff, MixOn
INPULSES heatAck
OUTPULSES mixAck
OUTPORT MixerActions
    MESSAGES MixerStart, MixerStop
MODE normal



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



In the following we give some pulse cast chains that can occur in the *ChainedPulseCast* System. We use the following notation:

```
anInpulse [aProcess] anOutpulse
                              denotes a process reaction (message output is not shown)

anOutpulse •
    anInpulse1
    anInpulse2                denotes a pulse cast, that is,
                                      anOutpulse causes anInpulse1 and anInpulse2
```

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]
```

Entering the *MixOn* message leads to

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

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]
```

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 tree, and the Operator process two times.

3.6 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 inpulse. 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.

3.6.1 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 26.

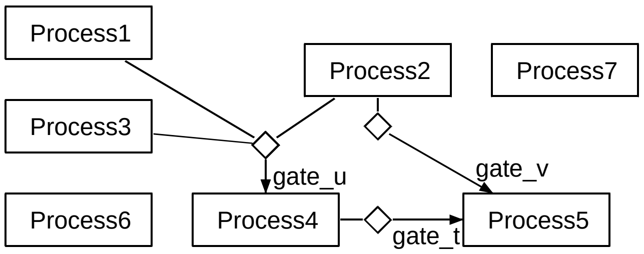


Figure 26: INSPECTION NET – Processes linked with inspection connectors

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 27 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 3.6.4.

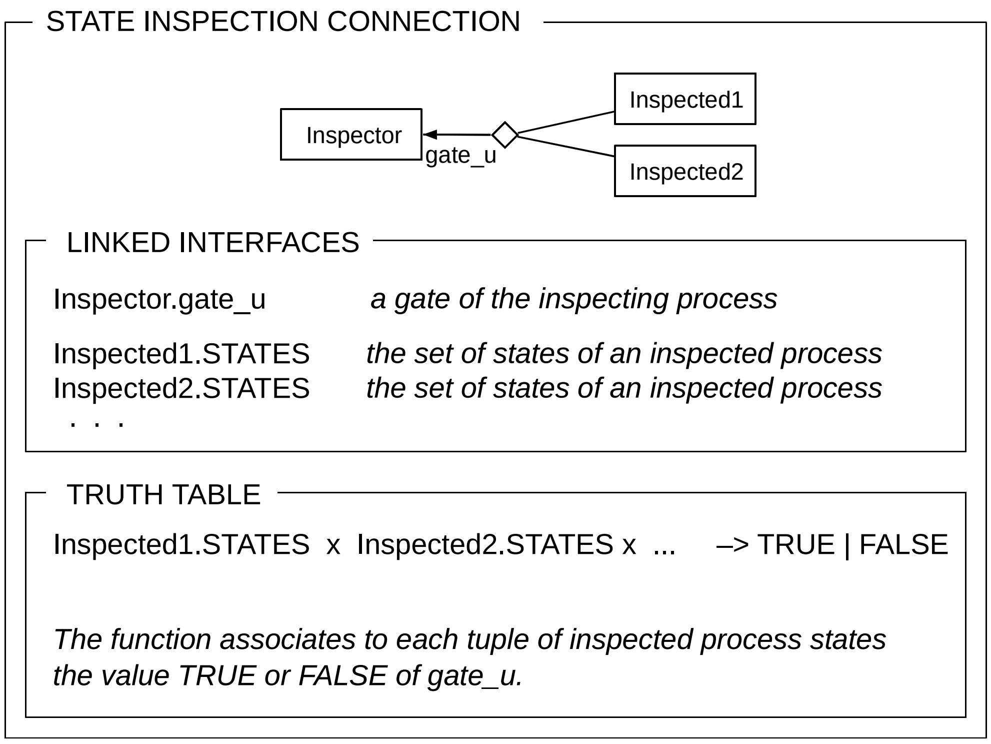

STATE INSPECTION CONNECTION
Inspector
gate_u
Inspected1
Inspected2
LINKED INTERFACES
Inspector.gate_u
a gate of the inspecting process
Inspected1.STATES
the set of states of an inspected process
Inspected2.STATES
the set of states of an inspected process
. . .
TRUTH TABLE
Inspected1.STATES x Inspected2.STATES x ... –> TRUE | FALSE
The function associates to each tuple of inspected process states the value TRUE or FALSE of gate_u.

Figure 27: Elements of a state inspection connection

3.6.2 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 3.3.2). 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.

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.

3.6.3 A Simple State Inspection Modelling Example

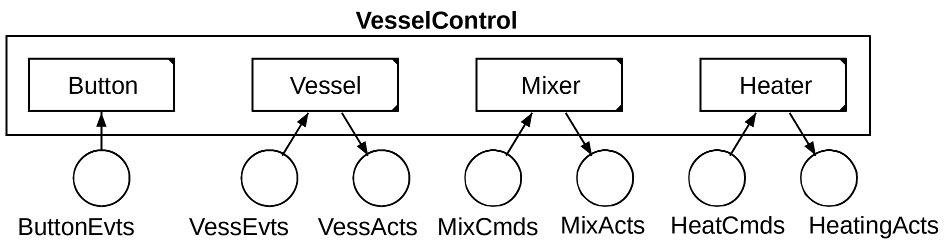
A vessel with an incorporated mixer and heater can be filled and emptied by pressing a button. The mixer and the heater are controlled by operator commands; the mixer can turn at low and at high speed. To empty the filled vessel, the heater must be turned off, and the mixer must not turn at high speed.

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.

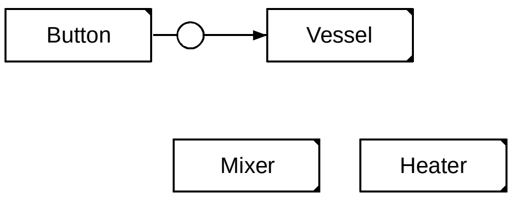
SYSTEM SimpleStateInspection

COMMUNICATION NET



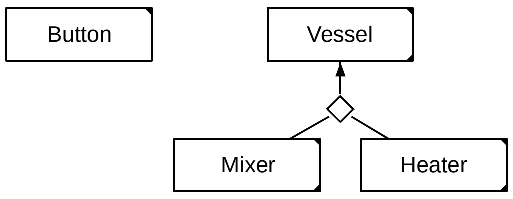
CLUSTER LampCluster

PULSE CAST NET



PULSE TRANSLATIONS
SENDER Button
press -> Vessel.press

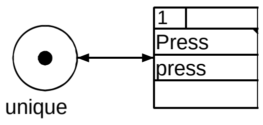
INSPECTION NET



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

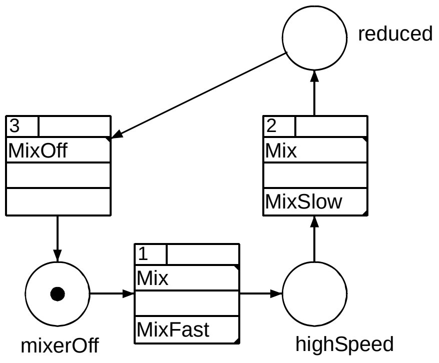
PROCESS Button
INPORT EventPort
MESSAGES Press
OUTPULSES press

MODE normal



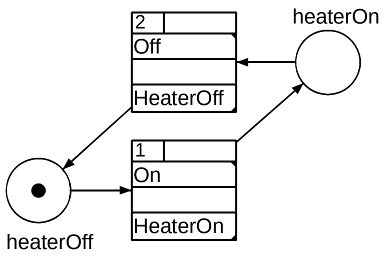
PROCESS Mixer
INPORT CmdPort
MESSAGES Mix, MixOff
OUTPORT ActionPort
MESSAGES MixFast, MixOff, MixSlow

MODE normal



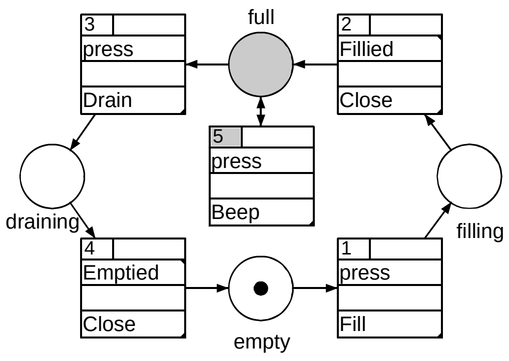
PROCESS Heater
INPORT CmdPort
MESSAGES Off, On
OUTPORT ActionPort
MESSAGES HeaterOff, HeaterOn

MODE normal



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_

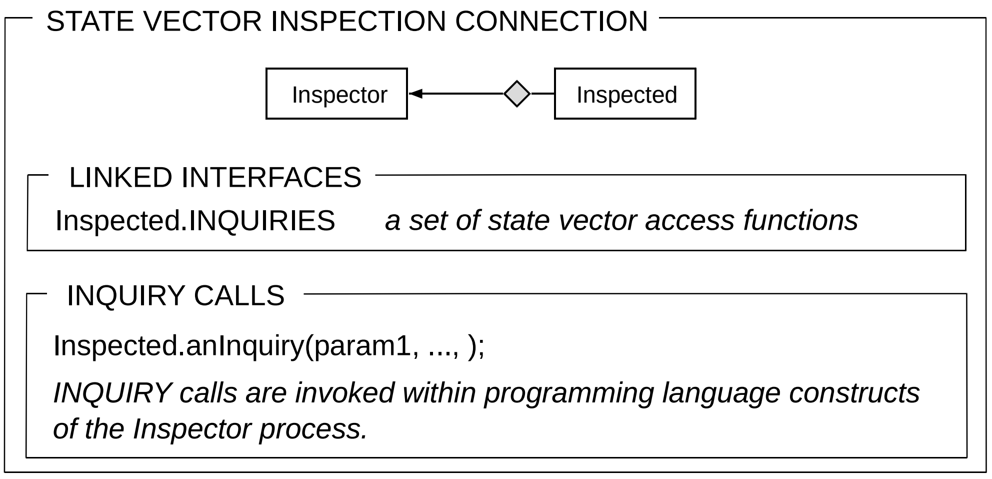
When the *Vessel* process is triggered in the *full* state by a *press* pulse, two state transitions are possible. When the *enableDraining* gate is true, the transition 3 to the *draining* state causes emptying the vessel; otherwise the transition 5 back to the *full* state causes a *Beep* signal.

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.

3.6.4 Extended State Inspection: State Vector Access by Inquiries

State dependencies based on the discrete states of processes (extended finite state machines) are modelled by *state inspection interaction*. The specification of dependencies based on process variables is correspondingly supported by a state vector access mechanism termed *state vector inspection*.

The frame of Figure 28 shows the elements of a *state vector inspection* connection. Connections are created 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 is a link from an *Inspected* process to an *Inspector* Process.



Inspected.INQUIRIES *a set of state vector access functions*

Inspected.anInquiry(param1, ...,);

INQUIRY calls are invoked within programming language constructs of the Inspector process.

Figure 28: The elements of a state vector inspection connection

The *Inspector* process can retrieve information of the current state vector of the *Inspected* process by calling *Inquiries* (state vector access functions) of the *Inspected* process. The call is formulated as
`Inspected.anInquiry();`
An *Inquiry* function always returns an integer value. In order to return values with other data types, pointer type function parameters have to be used. *Inquiries* can be called in any programming language construct (condition, operation, inquiry, ...) of an inspecting process.

3.6.5 A Simple State Vector Inspection Modelling Example

A computer controls a mixer and a heater. The action parameters of the control system can be updated during run time by entering a configuration message.

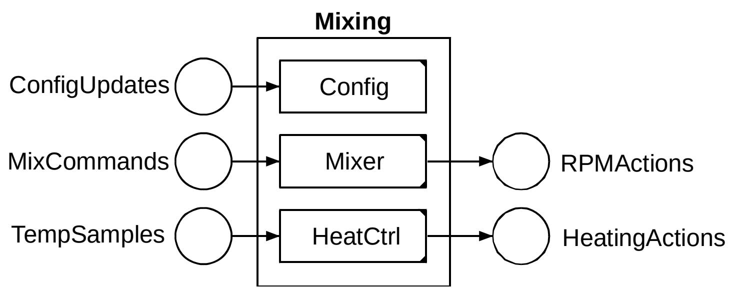
In the *SimpleStateVectorInspection* model below, new parameter values can be entered with the *Parameter* message of the *Config* process. The entered values are stored in the state vector variables of this process. The *RPM_Mixer* variable holds the rotation rate per minute of the mixer, whereas the *tempLevel* variable retains the temperature level below which the heater is to be turned off.

In addition, there is the *TEMP_DEFAULT* constant specified defining a default temperature level value. The default constant is defined in the INCLUDE text of the *Config* process. The *SetDefault* and the *Tuning* input messages of the *Config* process determine which temperature level is to be used.

The current control parameters are accessed by the *Mixer* and the *HeatCtrl* processes by calling the the appropriate Inquiry. The *RPM_Mixer* inquiry is called in the *setRPM* operation of the *Mixer* process, and the *TempLevel* inquiry is called in the *tempLevelReached* condition of *HeatCtrl* process.

SYSTEM SimpleStateVectorInspection (using direct inquiry invocations)

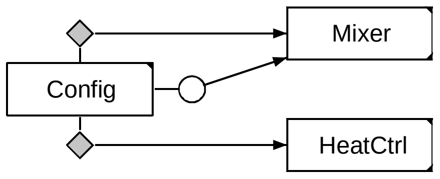
COMMUNICATION NET



USER TYPE tFloat
```
typedef float tFloat;
```
USER TYPE tInt
```
typedef int tInt;
```
USER TYPE tConfig
```
typedef struct{
  tFloat newRPM;
  tInt newTempLevel;
}tConfig;
```

CLUSTER Mixing

PULSE CAST & INSPECTION NET



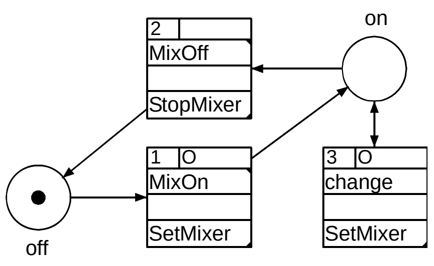
PULSE TRANSLATIONS

SENDER Config
change -> Mixer.change

PROCESS Mixer
INPORT CmdPort
MESSAGES MixOff, MixOn
OUTPORT ActPort
MESSAGES
SetMixer: tFloat, StopMixer

OPERATION setRPM
```
{ Config.RPM_Mixer(&OUTMSG;}
```

MODE normal



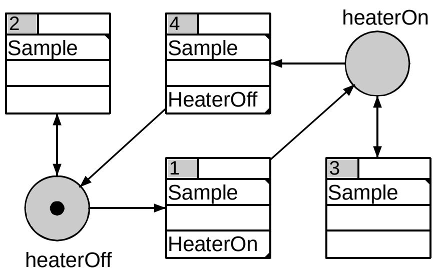
OPERATION ASSIGNEMENT
TRANSITION 1/
OUTPORT ActPort OPERATION setRPM

TRANSITION 3/
OUTPORT ActPort OPERATION setRPM

PROCESS HeatCtrl
INPORT Sampling
MESSAGES Sample: tInt
OUTPORT ActionPort
MESSAGES HeaterOff, HeaterOn

CONDITION tempLevelReached
```
IN > Config.TempLevel()
```

MODE normal



SWITCHES
STATE heaterOn MESSAGE Sample
TRANSITION 4 / tempLevelReached
TRANSITION 3 / ELSE_

STATE heaterOff MESSAGE Sample
TRANSITION 2 / tempLevelReached
TRANSITION 1 / ELSE_

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

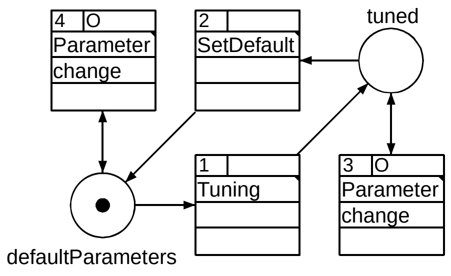
PROCESS Config

INCLUDE
```
#define TEMP_DEFAULT 80
```
INQUIRY TempLevel
```
int TempLevel (void){
    if (STATUS.STATE == tuned)
        return STATUS.tempLevel;
    else
      return TEMP_DEFAULT;
}
```
INQUIRY RPM_Mixer
```
int RPM_Mixer (tFloat* rpm){
    *rpm = STATUS.RPM_Mixer;
    return 1;
}
```
VARIABLES
RPM_Mixer: tFloat, tempLevel: tInt

INPORT ConfigPort
MESSAGES Parameter: tConfig, SetDefault, Tuning

OUTPULSES change

OPERATION getNewConfig
```
{
  SELF.RPM_Mixer = IN.newRPM;
  SELF.tempLevel = IN.newTempLevel;
}
```
MODE normal



OPERATION ASSIGNEMENT

TRANSITION 3 / OPERATION getNewConfig

TRANSITION 4 / OPERATION getNewConfig

3.6.6 State Vector Access by Answers linked to Queries

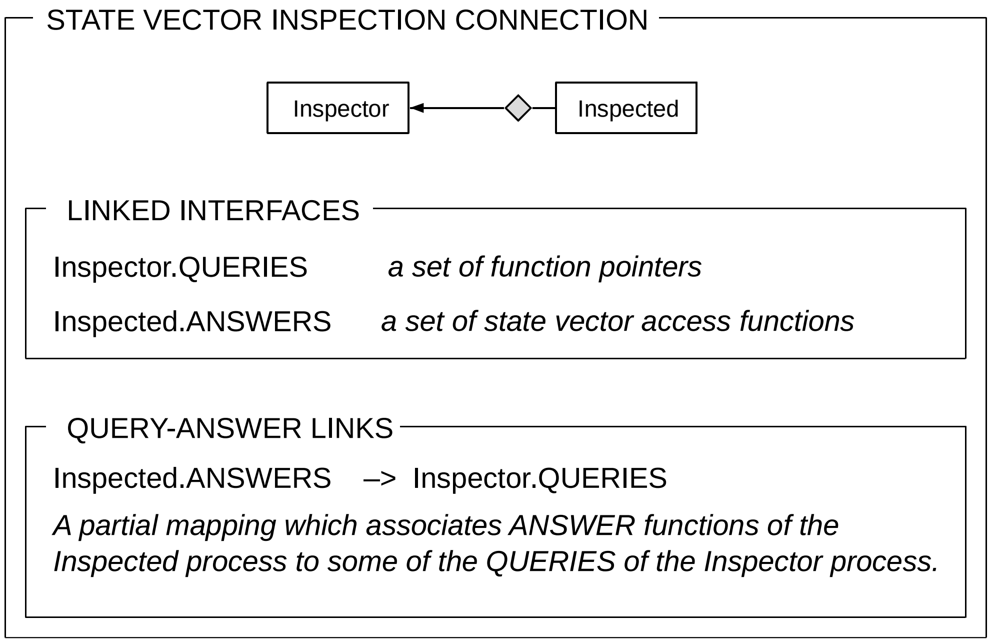
Inquiries represent elements of provided interfaces of inspected processes, like public methods of objects in OO-programs. These provided interface functions are called directly by inspecting processes, violating thereby the component paradigma. Components interact always via two interfaces linked by a connector.

In order to spport the component paradigma for state vector inspection interaction too, inspecting processes must declare a required inspection interface, and the links from required to provided interfaces must be specified as a property of the corresponding state vector connnection.

A State Vector Inspection Mechanism Supporting the Component Paradigma

(not yet supported by CIP Tool)

The frame of Figure 29 shows all elements of a state vector inspection connection based on linked interfaces. The two linked interfaces consist of a set of *Queries* of the inspecting process and a set of *Answers* of the inspected process. A *Query* of a process represents an element of the required inspection interface and is specied by function pointer described in the programming language of the extended state machine. An *Answer* is, like an *Inquiry,* an implemented state vector access function. The set of *Query-Inqury* links of a connection is a mapping that defines how *Queries* of the inspecting process are linked to *Answers* of the inspected process. The mapping must therefore relate *Answers* to *Queries* of the same function type. The same *Answer* can be linked to *Queries* of several inspecting processes, whereas each *Query* must have a uniquely linked Answer defined by one particular inspection connection.



Inspector.QUERIES *a set of function pointers*

Inspected.ANSWERS *a set of state vector access functions*

Inspected.ANSWERS –> Inspector.QUERIES

A partial mapping which associates ANSWER functions of the Inspected process to some of the QUERIES of the Inspector process.

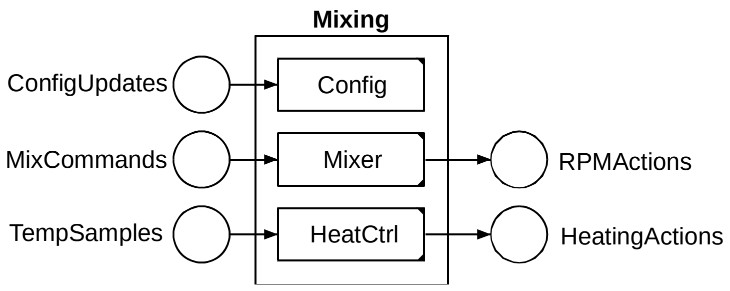
Figure 29: The elements of a state vector inspection connection

When an inspecting process calls a query the linked answer of the inspected process is invoked.

On the following page, the *SimpleStateVectorInspection* System is specified using *Answers* and *Queries*. The *Config* Process is not given. It is the same as in the previous example, except for the *Inquiries*, which are defined now as *Answers*.

SYSTEM SimpleStateVectorInspectionUsingAnswersAndQueries

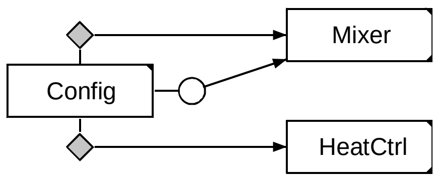
COMMUNICATION NET



USER TYPE tFloat
```
typedef float tFloat;
```
USER TYPE tInt
```
typedef int tInt;
```
USER TYPE tConfig
```
typedef struct{
  tFloat newRPM;
  tInt newTempLevel;
}tConfig;
```

CLUSTER Mixing

PULSE CAST & INSPECTION NET



PULSE TRANSLATIONS
SENDER Config
change -> Mixer.change

QUERY-ANSWER LINKS

Mixer.getRPM -> Config.RPM_Mixer

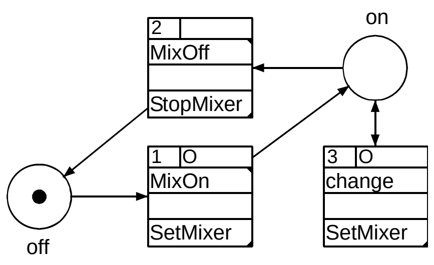
HeatCtrl.getTempLevel -> Config.TempLevel

PROCESS Mixer

OUERY getRPM
```
tFloat (*getRPM)(void)
```
INPORT CmdPort
  MESSAGES MixOff, MixOn

OUTPORT ActPort
  MESSAGES
    SetMixer: tFloat, StopMixer

OPERATION setRPM
```
{OUTMSG = QUERY.getRPM();}
```
MODE normal



OPERATION ASSIGNEMENT
TRANSITION 1/
OUTPORT ActPort OPERATION setRPM

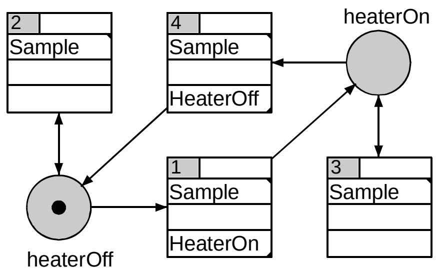
TRANSITION 3/
OUTPORT ActPort OPERATION setRPM

PROCESS HeatCtrl

OUERY getTempLevel
```
tInt (*getTempLevel)(void)
```
INPORT Sampling
  MESSAGES Sample: tInt

OUTPORT ActionPort
  MESSAGES HeaterOff, HeaterOn

CONDITION tempLevelReached
```
IN > QUERY.getTempLevel()
```
MODE normal



SWITCHES
STATE heaterOn MESSAGE Sample
TRANSITION 4 / tempLevelReached
TRANSITION 3 / ELSE_

STATE heaterOff MESSAGE Sample
TRANSITION 2 / tempLevelReached
TRANSITION 1 / ELSE_

3.7 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.

3.7.1 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 30 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.

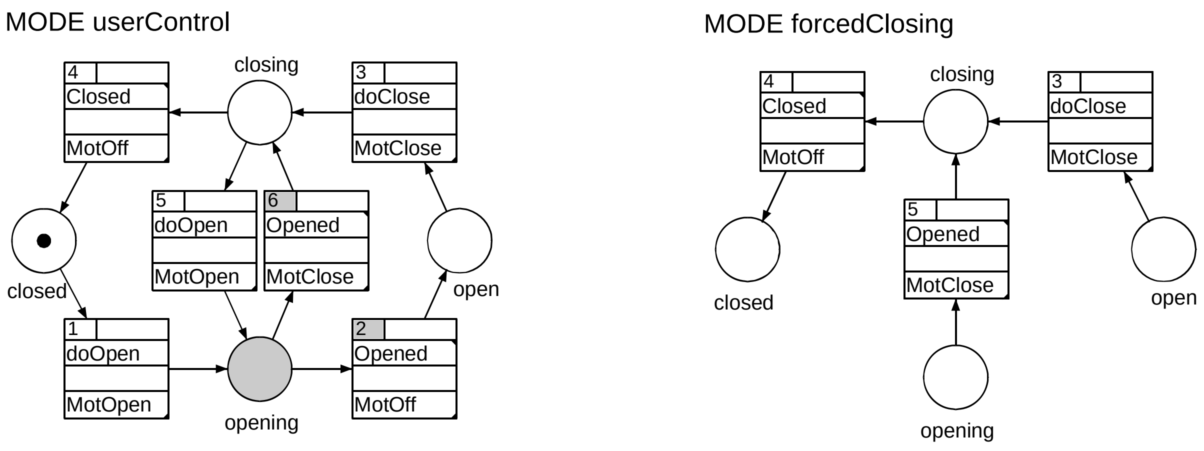


Figure 30: Two modes of a door control process

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.

3.7.2 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 31. 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.

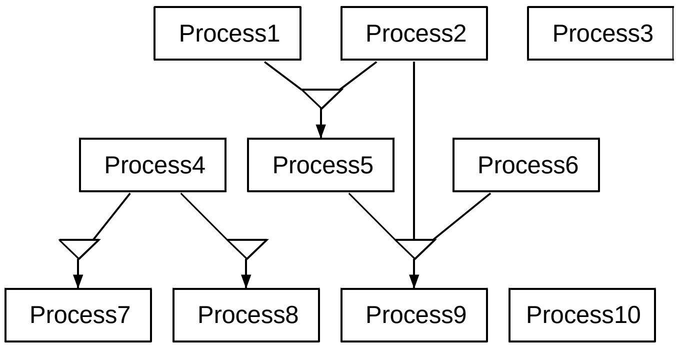


Figure 31: MODE CONTROL NET – Processes linked with mode control connectors

The levels indicated in the graph of Figure 31 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.

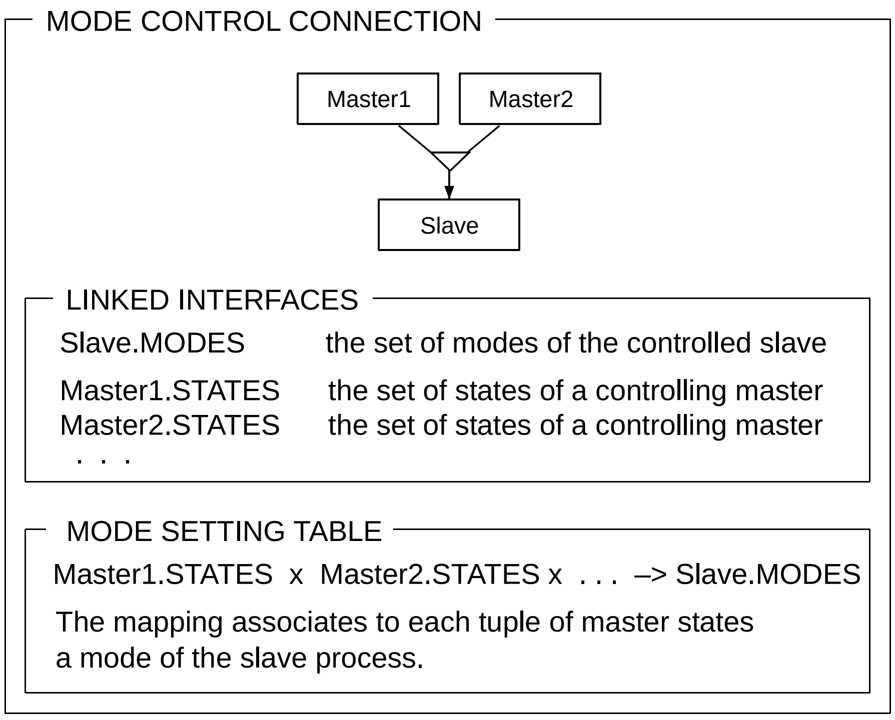

MODE CONTROL CONNECTION
Master1
Master2
Slave
LINKED INTERFACES
Slave.MODES the set of modes of the controlled slave
Master1.STATES the set of states of a controlling master
Master2.STATES the set of states of a controlling master
. . .
MODE SETTING TABLE
Master1.STATES x Master2.STATES x . . . –> Slave.MODES
The mapping associates to each tuple of master states a mode of the slave process.

Figure 32: The elements of a mode control connection

The frame of Figure 32 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.

3.7.3 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 30 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.

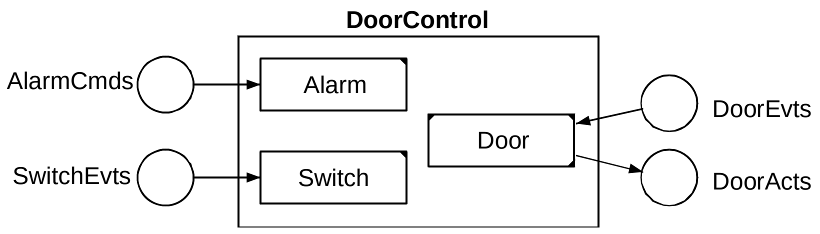
It is important to note that a change of mode does not affect the current state of a slave; a state can change only when a transition in the active mode is triggered by a message or a pulse. The stability of state with respect to mode change is an essential property of mode control interaction. The current 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.

3.7.4 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.

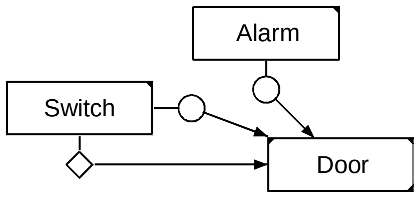
SYSTEM SimpleModeControl

COMMUNICATION NET



CLUSTER LampCluster

PULSE CAST & INSPECTION NET



PULSE TRANSLATIONS

SENDER Alarm
    doClose -> Door.doClose

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

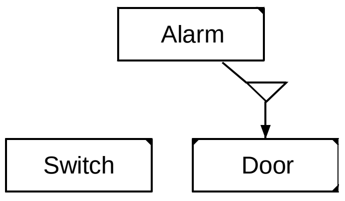
TRUTH TABLE

INSPECTOR Door GATE holdOpen

RESPONDERS Switch

TRUE <- down

MODE CONTROL GRAPH

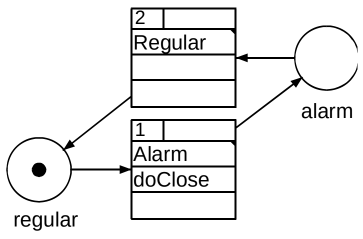


MODE SETTING

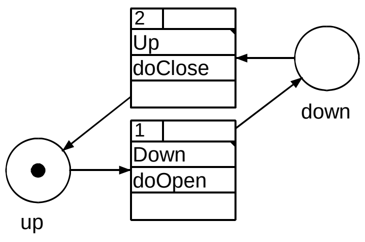
SLAVE Door

MASTERS Alarm
    forcedClosing <- alarm
    userControl <- regular

PROCESS Alarm
INPORT CmdPort
  MESSAGES Alarm, Regular
OUTPULSES doClose
MODE normal

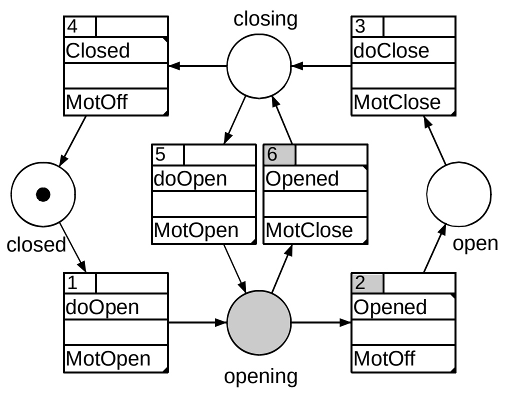


PROCESS Switch
INPORT EventPort
  MESSAGES Down, Up
OUTPULSES doClose, doOpen
MODE normal

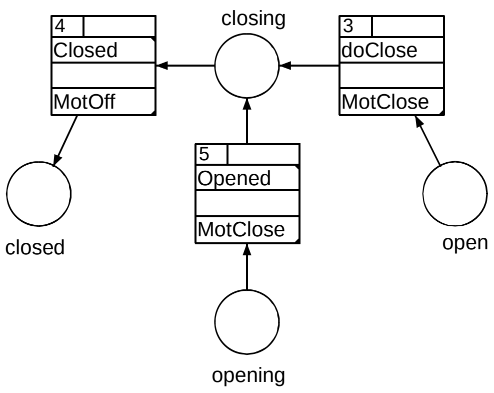


PROCESS Door
INPORT PositionEvents
  MESSAGES Closed, Opened
INPULSES doClose, doOpen
OUTPORT MotorActions
  MESSAGES MotClose, MotOff, MotOpen
GATE holdOpen

MODE userControl



MODE forcedClosing



SWITCH
STATE opening MESSAGE Opened
TRANSITION 2/ 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*.

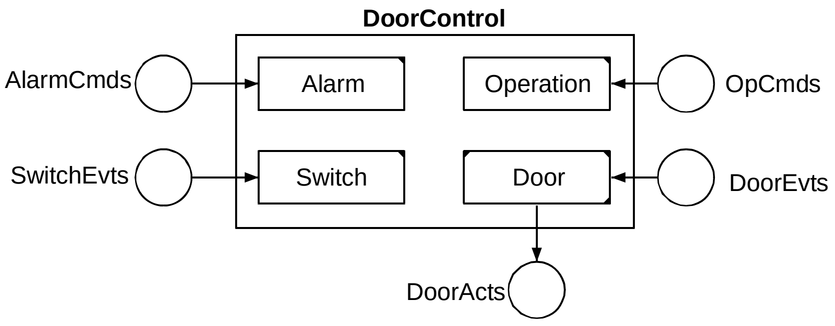
Note that an alarm may be released immediately after commanding the alarm. In this case it is possible that after releasing the alarm the button is pressed to reopen the door before the door has completely been closed.

3.7.5 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.

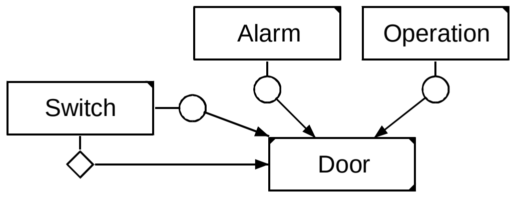
SYSTEM ExtendedModeControl

COMMUNICATION NET



CLUSTER LampCluster

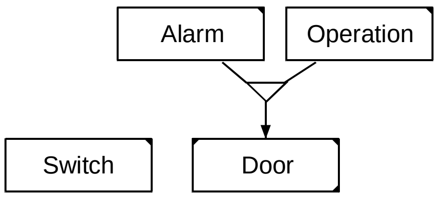
PULSE CAST & INSPECTION NET



TRUTH TABLE
INSPECTOR Door GATE holdOpen
RESPONDER Switch
TRUE <- down

```
PULSE TRANSLATIONS
SENDER Alarm
   trig          -> Door.trig
SENDER Operation
   trig          -> Door.trig
SENDER Switch
   doClose       -> Door.doClose
   doOpen        -> Door.doOpen
```

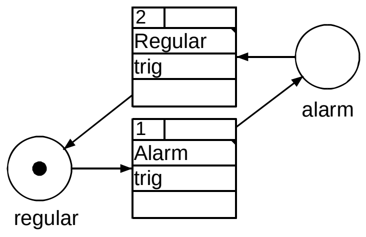
MODE CONTROL NET



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

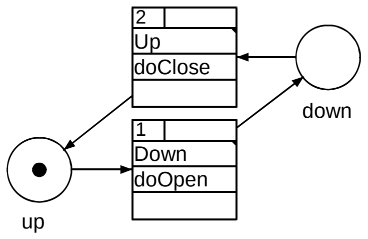
PROCESS Alarm

MODE normal



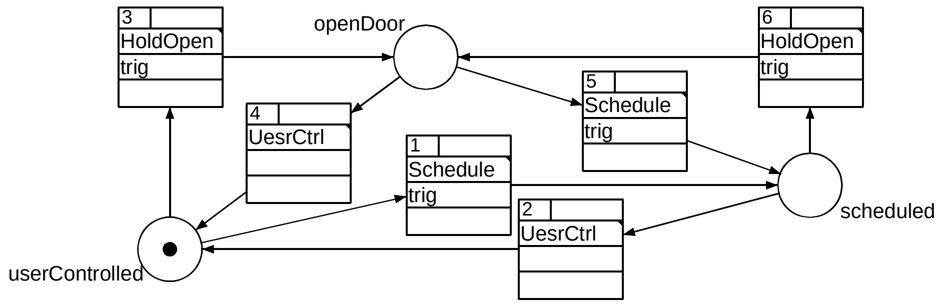
PROCESS Switch

MODE normal



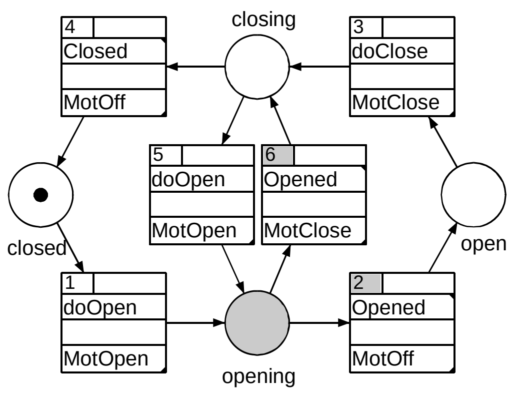
PROCESS Operation

MODE normal

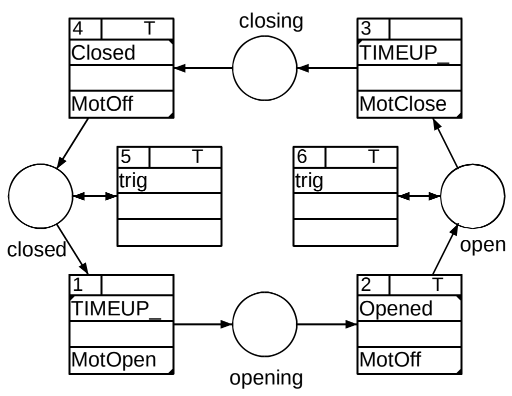


PROCESS Door

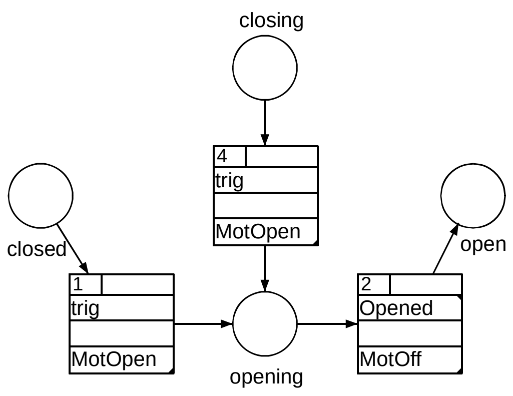
MODE userControl



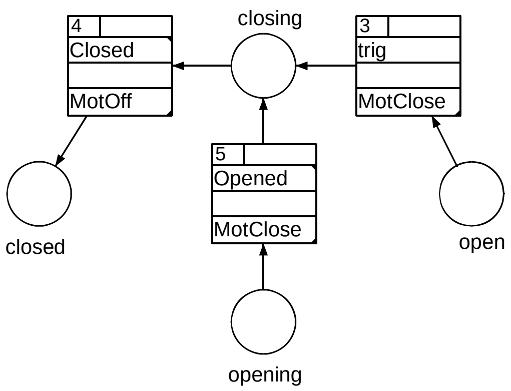
MODE scheduled



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 a technical 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 technical 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. 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.

3.8 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.

3.8.1 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 inpulse).

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

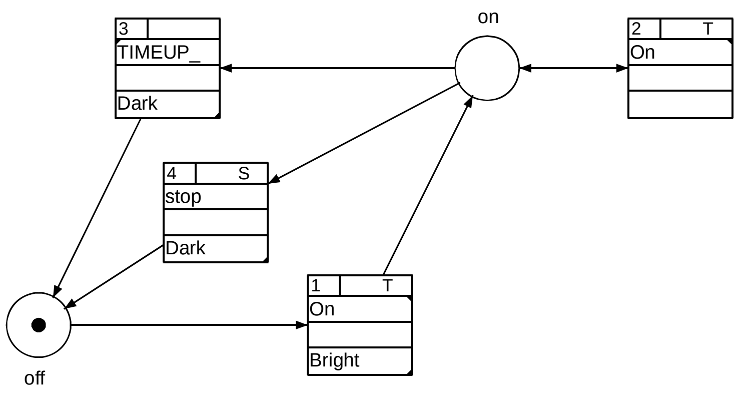
INPULSES stop

OUTPORT ActPort
  MESSAGES Bright, Dark

DELAYS

DELAY set5sec
```
5 * TICKS_PER_SECOND
```
DELAY set10sec
```
10 * TICKS_PER_SECOND
```

MODE Normal



OPERATION ASSIGNEMENT

TRANSITION 1 / DELAY set5sec

TRANSITION 2 / DELAY set10sec

Figure 33:

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

PROCESS ChainedExeptionManager
MODE normal

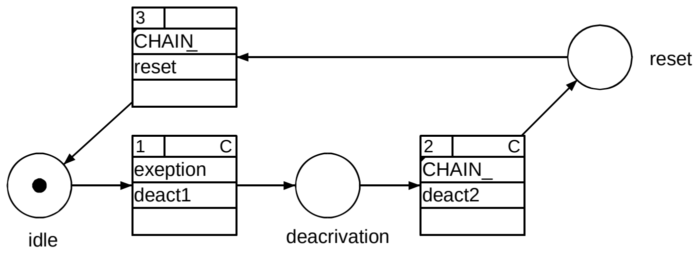


Figure 34:

3.8.3 AUTO-EXTENSION

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

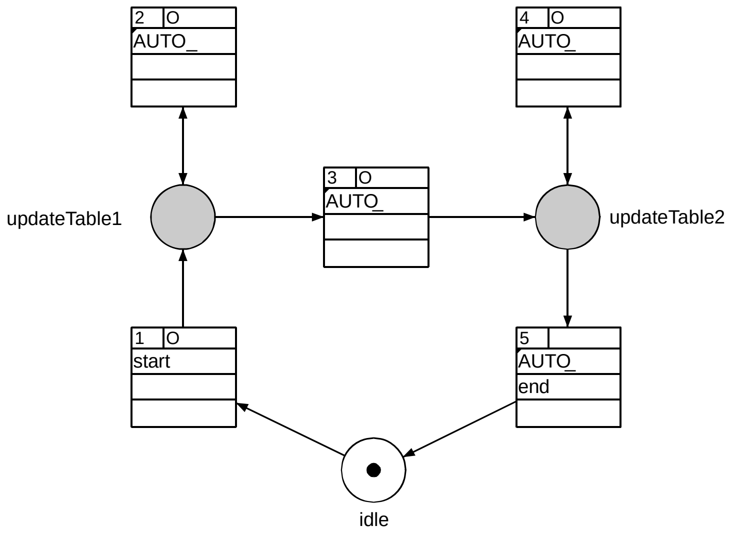
> **AUTO_**
> The AUTO_ trigger can be assigned to any state transition of the process as input stimulus. AUTO_ can occur at any time. An unexpected AUTO_ trigger is ignored.

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


2
O
AUTO_
4
O
AUTO_
3
O
AUTO_
updateTable1
updateTable2
1
O
start
5
AUTO_
end
idle