Steam Boiler Control Specification Problem CIP Solution

Hugo Fierz
Institut für Technische Informatik und Kommunikationsnetze, ETH-Zentrum
CH-8092 Zürich, Switzerland, E-mail: fierz@tik.ee.ethz.ch

Abstract. CIP (Communicating Interacting Processes) is a formal development method for specifying reactive systems by means of cooperating extended state machines. The methodological approach starts out from the behaviour of the objects of the environment, leading in constructive steps to an executable specification of the reactive system. This paper presents a complete CIP solution of the Dagstuhl Steam Boiler Control Problem. The executable CIP specification is given as a generated report from CIP Tool, a tailor-made software tool supporting an efficient application of the CIP method in practice.

This paper presents a CIP solution of the steam boiler control problem [1]. The control system has been specified using *CIP Tool®* [3], a software tool consisting of a system of graphical and text editors, C-code generators and a report writer. From the complete and consistent system specification (automatic verification) we generated executable C-code and linked it with a MATLAB/Simulink® model [4] of the steam boiler in order to validate the behaviour of the control system by simulation. The produced C-code is independent of any run time library or operating system. Several industrial applications developed with CIP have proven that the generated code can be used without modification on the target system. The reason for the sufficient performance lies in the nature of the CIP specification model which forces the developer to make implicit design decisions (architecture, control flow) already in the functional specification.

The presented CIP solution of the steam boiler problem is explained within the comments of a generated system report. The report gives a formally complete description of the specified steam boiler controller.

The future development of the CIP method concerns the formal verification of temporal properties of system specifications, described as temporal correlations of subsets of process states. The formal basis of CIP will then allow to support automatic verification of such properties.

Overview. We start by briefly describing the specification formalism and the development concepts of the CIP method. In Section 2 the meaning of the objects of a CIP specification is explained within the comments of a generated report of a dummy specification. Section 3 explains the CIP problem domain of the steam boiler control system. Section 4 contains the result of the first development phase of the steam boiler control system: Events, actions and model processes. Section 5 presents the complete CIP specification of the steam boiler control system as generated system report.

1. Introduction to CIP – Communicating Interacting Processes

CIP is a formal development method for distributed reactive systems [5, 6]. A CIP specification describes by means of cooperating state machines a working solution of the developed system.

Description technique and development concepts of the CIP method

- Operational specification

The behaviour of a reactive system is formally described by means of cooperating state machines.

- Visual formalism

Behaviour of processes, data flow and control flow are specified by means of graphic models.

- Environment oriented development approach

The methodological approach starts out from the behaviour of the objects of the environment, leading in constructive steps to the operational specification of the reactive system.

1.1. Operational specification of reactive systems

An *operational specification* [7] is the formal description of the behaviour of a system by an abstract machine (Petri nets [8], state machines [9], sequential processes [10]. *Operational* signifies that the basic model is logically executable. Suitable software tools enable executable codes to be generated from such models. Documentation with the same formal content as the generated code can also be generated from the same models.

In principle two different problems have to be solved in developing a reactive system:

- Logically correct reaction to environmental events (reactive layer)
- Transmission of information between reactive system and environment (transmission layer)

As far as possible these two problems should be solved independently. The solution to the logical problem can be regarded as the invariant part of different implementations of a reactive system and is accordingly described in a higher abstraction level as abstract machine. The logical description of how, for example, the control of a lift system behaves, will not contain any terms used in computer technology. However, we can only talk of something being a solution to the logical problem when the solution is formally described. On the other hand, the problem of information transfer is strongly influenced by the hardware and has to be solved using appropriately specialised techniques.

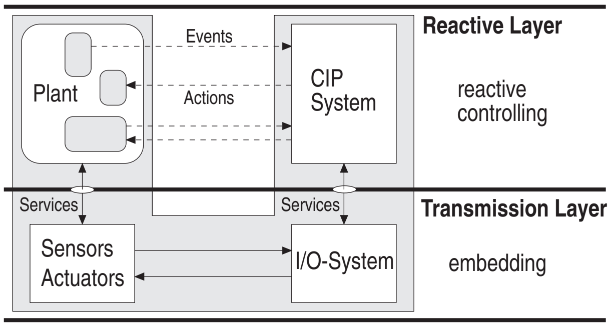


Fig. 1 Abstraction levels of reactive systems

The complete separation of the logical problem solution from the implementation of the information transmission considerably benefits the development process since the two problems can be solved independently of each other. This is not just a question of reducing a large problem to two smaller ones but the unravelling of two problem complexes belonging to different abstraction levels.

1.2. The operational specification formalism of CIP

In the CIP method, the reactive system behaviour is specified by cooperating state machines. A CIP system comprises asynchronous cooperating clusters (true concurrency). Each cluster is composed of synchronously cooperating processes which are described as extended state machines. The mathematical model of a cluster is defined as a product of these state machines and represents a state machine with multi-dimensional state space, i.e. the state of a cluster is given by the states of its processes.

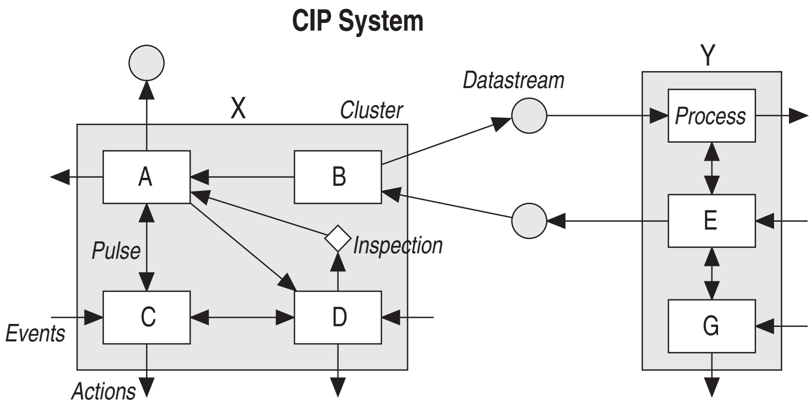


Fig. 2 CIP system with two clusters

An ocurring event of the environment leads to a state transition of the receiving process. By means of a broadcasted pulse, the process thus activated can activate further cluster processes which can in turn activate other processes by pulses. The chain reaction resulting from pulse transmission can not be interrupted and defines a single state transition of the entire cluster. The activated processes react on the environment by means of actions.
The processes of the different clusters are capable of asynchronous communication via datastreams. In the CIP formalism, asynchronous communication means that the write and read actions of a transmission are executed in chronological separation in two different cluster transitions. Datastreams may of course also be used to communicate between processes of the same cluster.

Processes – Extended state machines

Processes are specified as extended state machines. With transition structures and by means of operations executed in the transitions, functionality can be specified in two different abstraction levels. The transition structure of a process describes the cooperation with other processes and with objects in the environment; with operations one defines the local algorithmic functionality.
The transition structure depicted in Fig. 3 specifies the behaviour of the process *Casher* of a vending machine. The process is always in one of the states represented by cercles. The external event *Coin* or the pulses *abort* and *served* can initiate a state transition. The action *Eject* and a new pulse can be generated in each transition.

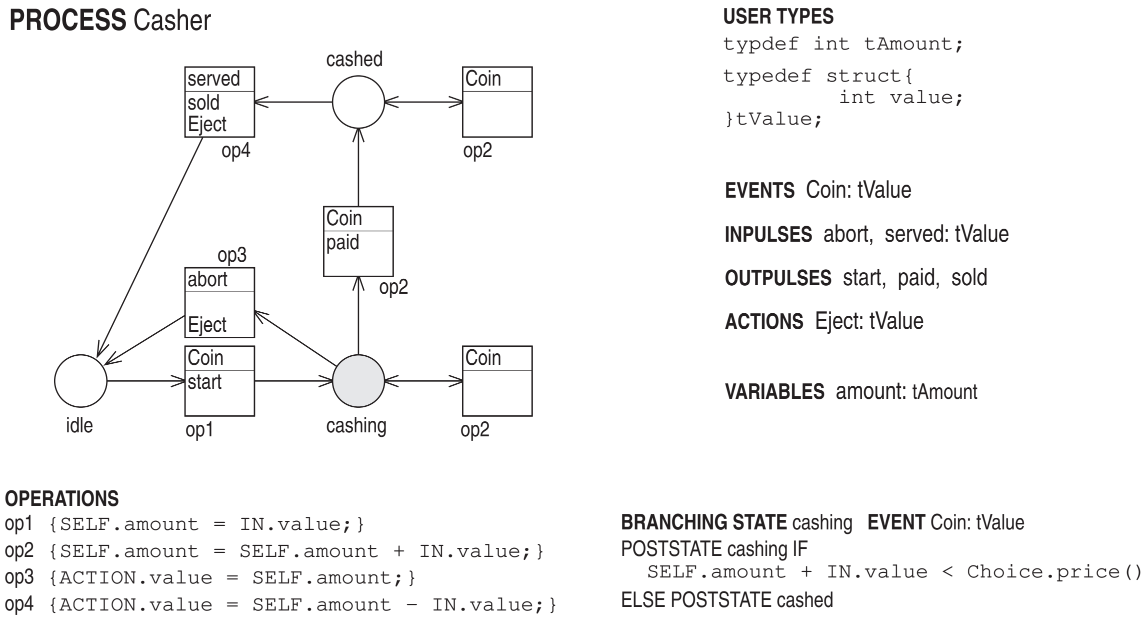


Fig. 3 The process *Casher* of a vending machine

For algorithmic concerns static variables and data types can be declared per process for events, pulses, actions and messages. The algorithms are specified by means of operations executed in the state transitions.
When the process Casher is in the cashing state - shown in grey - Fig. 3, there are two potential transitions for the event Coin (non-determinism). Corresponding conditions have to be assigned to the two transitions in order to render the process behaviour unique. The conditions of such a branching can depend on the event data, the values of the local process variables and also on the states and variables of other processes of the same cluster (see state inspection).
Variables, data types, operations and conditions are formulated in CIP in the programming language C or C++. The specified code fragments are incorporated «inline» in the generated C-code. From the theoretical aspect it would be more elegant to use an inherent functional language. In practice however the usefulness of the pragmatic approach with ANSI-C, resp. C++ has been clearly confirmed since it permits easely the use of functions, data types and object classes from existing libraries.
As a general rule, reactive systems are of a hybrid character with the acquisition of information from the environment taking place partly via discrete signals and partly by quasi-continuous sampling. The control of sampled real objects is specified in CIP using data-carrying sampling events and feedback actions; the control algorithms are executed in operations of the activated transitions of the corresponding CIP process.

Interaction – Synchronous pulse transmission, state inspection

The synchronous transmission of pulses is described as interaction. An *interaction net* (Fig. 4) is used to specify between which processes pulses can be transmitted.

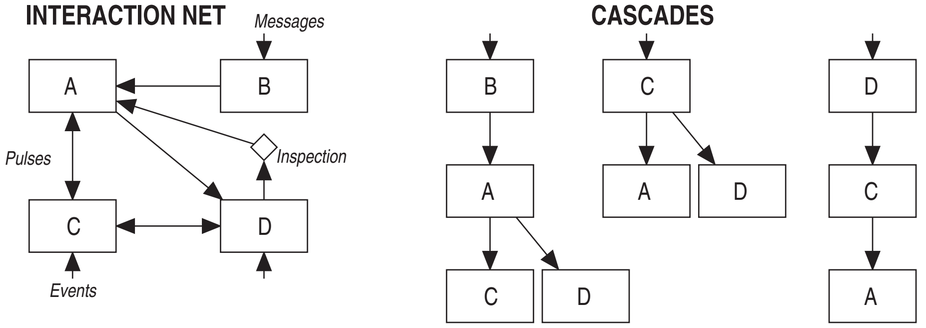


Fig. 4 Interaction net and cascades of a cluster

However, the structure of the interaction net does not sufficiently restrict the potential pulse transmission chains as cyclic transmission paths are generally possible. In order to ensure bounded and unique chain reactions, for every process with external input the possible control flow is specified by a corresponding process cascade. A *cascade* is a tree-form subnet of the interaction net, thus every process appears at most once in a cascade. A cascade is always activated by an external input for its topmost process. If a process activated in a cascade generates no pulse, or a receiver does not await a generated pulse, the execution of the cascade is broken off in this node.

As already mentioned, the conditions of a transition structure can be directly dependent on the states and variables of other processes in the same cluster. The read access to the data of a process is called a state inspection and takes place like in object oriented models via access functions (inquiries) of the process being inspected. Process Casher of Fig. 3 contains an example of an inquiry call: *Choice.price()* in the branching condition for poststate cashing. The information supplied always refers to the states and variable values immediately prior to the current cluster activation. By contrast to pulse transmission, where both transmitter and receiver are activated, in the case of an inspection, only the inspecting process is active.

State inspection gives rise to additional dependencies between processes which are shown graphically as diamonds in the interaction net (Fig. 4). The arrows denote the data flow direction.

Communication – Asynchronous transmission of messages

The information transmission between processes of different clusters takes place by asynchronous communication by means of datastreams. Datastreams are sequential message buffers and model the communication medium of the distributed system. As with interaction, the dataflow structure is specified by a graphic net model, the communicating processes being linked with datastreams via inports and outports. Outport and inport messages can be written and read in process transitions.

Moderation – Behaviour structuring

In order to support the specifcation of different behaviour patterns for a process CIP alows for the definition of several *modes* for a process. Each mode is described by a transition structure as in Fig. 3 which uses the states, events, pulses, actions and messages of the process.

The active mode of a process defines the actual process behaviour. A further transition structure on the process modes (*moderator*) is used to specify how the process behaviour can be changed. Mode transitions are activated by special pulses. With a mode transition, the current process state is retained; it is possible however to relocate a trigger in a mode transition which, in the new mode, can immediately lead to a first state transition.

The structuring of the process behaviour by different modes is useful for separating the direct influencing of a process from its behavioural influences. This means of structuring provides a simple way of mastering the complexity of reactive systems using behaviour hierarchies.

Denotational Semantics of CIP

The formal semantics of synchronous cooperation is based on SCSM (Synchronous Composition of Sequential Machines) [11], a mathematical formalism developed for this purpose. SCSM allows to define synchrounous compositions of state machines. The meaning of a cluster specification is then defined by a semantic function which associates all CIP constructs of the specification with corresponding SCSM-expressions. The meaning of the asynchronously cooperating clusters is defined by a petri net model in which petri net state machines are coupled to petri nets modelling the datastreams [12].

1.3. Environment-oriented object-based development approach

With an operational specification formalisms one works automatically in a defined abstraction level. In addition to these "guard rails", a development method must also propose a conceptual model of the development process which enables intelligible solutions to be found.

CIP differs from most other methods in that the development of a functional specification starts with an environment model and postpones the definition of the functionality to a second phase. The approach originates from the JSD method (Jackson System Development) [10] and is also used in recent object-oriented analysis methods. The reality-oriented procedure leads to intelligible systems which have a direct influence on quality characteristics such as reliability, maintainability and modification-friendliness.

In this paper we describe the modelling and function phase of the CIP method applied to the development of a singular cluster. Complex systems consisting of several clusters would also require a development phase for global communication.

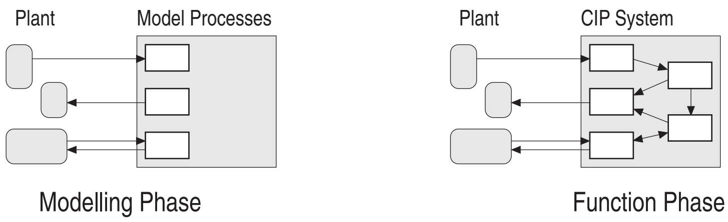


Fig. 5 Development phases of the CIP method

Modelling phase

In the modelling phase, the environment objects are modeled in terms of state machines. For every modeled real object a corresponding *model process* is specified which receives the sensor signals from the object as events and which can generate with actions appropriate actuator signals. Through the transition structures of these processes all possible sequences of occurring events and generated actions are specified. The provisionally incomplete model processes define in fact the interaction protocol for the reactive system and its environment. In this way the specified system contains explicit models which define the valid behaviour of the environment and thus form the basis for the development of the reactive functionality.

Function phase

The overall behaviour of the reactive system is specified in the second development phase. *Function processes* are introduced and the required dependencies between the model processes are defined by means of pulse transmission and state inspection. In a first step we develop the primary functionality of the system which is based on the normal behaviour as defined by the model processes. To also permit reaction to invalid behaviour of the environment and to transmission errors, it is usually necessary to extend the structures of the model processes and introduce supervisory processes in further steps. Past experience shows that the specification of system behaviour in the event of an error is one of the most serious problems in the development of robust and dependable systems.

2. Explaining CIP by means of a commented report of a dummy system

The following report of the dummy system *ReportHelp*, generated by *CIP Tool®*, eplains the meaning of the specified CIP objects. CIP object names of the dummy specification are of the form aCluster, aProcess, aState, etc.

The complete CIP specification of the Steam Boiler Controller is given in Section 5 as generated report. Thus *ReportHelp* may serve as help utility when the SteamBoilerController specification is analysed by the reader.

Structure of a generated CIP system report:

```
systemReport = {systemObject} {clusterReport}
clusterReport = {clusterObject} {processReport}
processReport = {processObject} {modeReport}
```

Meaning of report fonts:

KEYWORD	Helvetica capital
Definition of a CIP Object	Helvetica
Reference to CIP Object	Helvetica italic
`C-code`	Courier
Comment	Times

SYSTEM ReportHelp

A CIP system consists of several asynchronously cooperating clusters. A cluster is a multi-dimensional state machine. The several degrees of freedom are specified as interacting and communicating processes described by ordinary extended state machines.

INTEGER CONSTANTS
anintConstant: 20

FLOAT CONSTANTS
aFloatConstant: 23.55

CIP TYPE aCIPType
FIELDS aField1: INTEGER, aField2: *aUserType*

Record type used for process inputs, outputs and variables. Inputs and outputs without a CIP type carry no data.

USER TYPE aUserType
`typedef int aUserType;`

General ANSI-C data type definition used for process variables and CIP type fields.
(INTEGER, FLOAT and STRING are primitive CIP objects)

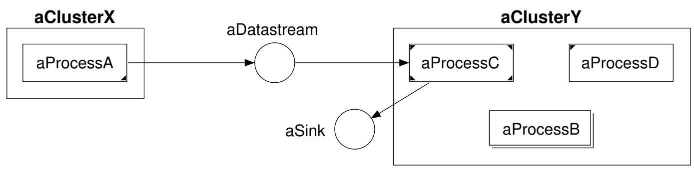
PORT TYPE aPortType
MESSAGES aMessage1: *aCIPType*, aMessage2

Message set used by process inports and outports serving as communication protocol.

INDEX TYPE anIndexType [*anIntConstant*]

An index type defines a one-dimensional finite multiplicity. Index types are used to specify multiplicities of processes and outports.

COMMUNICATION NET aCommunicationNet



Data flow diagram.
Processes communicate asynchronously by means of datastreams. Process aProcessB has many instances.

Legend for corner marks of process boxes:

Top left: receives events, TIMEUP or CHAIN	*Top right: reads messages*
Bottom left: generates actions	*Bottom right: writes messages*

DATASTREAM aDatastream [4]
READER *aProcessC.anInport* WRITER *aProcessA.anOutport*

DATASTREAM aSink [1]
WRITER *aProcessC.anInport*

Datastreams are sequential communication buffers (FIFO) with specified buffer size. Reader and Writers are listed with the corresponding connected port.

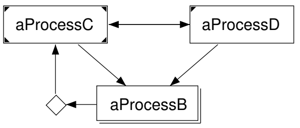
CLUSTER aClusterX

PROCESS aProcessA

OUTPORT anOutport: *aPortType*

CLUSTER aClusterY

INTERACTION NET

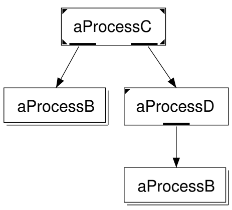


Data flow diagram.
The interaction net specifies the interaction data flow of a cluster. Senders and receivers of pulses are connected by straight directed lines. Diamond connections (direction = data flow) define state inspections enabling the inspector to call inquiries of the inspected process. For processes with multiple instances the pulse transmission on instance level is determined by the combination of the index types of the sender and the receiver.

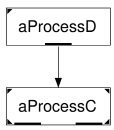
PULSE TRANSLATIONS
SENDER *aProcessC*
anOutpulse -> *aProcessB.anInpulse*
anOutpulse1 -> *aProcessD.anInpulse*
SENDER *aProcessD*
anOutpulse1 -> *aProcessB.anInpulse*, *aProcessC.anInpulse*
anOutpulse2 -> *aProcessC.aCommand*

Processes are specified as independent state machines with local input and output interfaces. Thus the pulse flow specification defined by the interaction net must be completed by explicitly defined pulse translation functions. Every outpulse of a sender is associated with receiver inpulses (or commands). Not translated outpulses are not transmitted.

CASCADE *aProcessC*



CASCADE *aProcessD*



Legend for gates:
The ordered set of gates of a process are represented as set of bars from the left to the right at the bottom of the process box.

Control flow diagrams.
Cascades are tree like subnets of the interaction net defining how pulse transmission is executed (control flow). A cascade is activated when its root process receives an external input (event, message, TIMEUP, CHAIN).

The outpulses of process aProcessD belong to one gate, the ones of aProcessC to two gates. For every gate of a process the subcascade below the gate defines the control flow activated by emitted pulses of the gate. Gates allow to specify branchings in the control flow of a cluster.

In an executed cascade every process can be activated at most once which restricts the possible structures for cascades (tool support).

PROCESS aProcessB

Scooping: Names of CIP objects of a process are local to the process.

INQUIRY anInquiry

```
int anInquiry (void)
{ return (STATUS.STATE == aState2 && STATUS.myVariable == 5); }
```

Inquires are access functions to the stored state vector of a process. The stored state vector of a process is only updated when the current cluster transition has terminated. Inquires can be called within conditions, operations, delays and inquiries of inspecting processes of the cluster.

The value of the state vector of a process consists of the actual mode (see below), of the current state and of the values of all process variables. The value of the mode, of the state and of a variable myVariable are referenced as `STATUS.MODE`, `STATUS.STATE` and `STATUS.myVariable` respectively.

An inquiry of a multiple process can access all instances of the process. The state vector of an instance is accessed as `STATUS[anIndex].myVariable`.

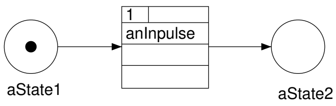
MULTIPLICITY *anIndexType*

The multiplicity of a process is defined by a set of associated index types.

VARIABLES myVariable: INTEGER

INPULSES anInpulse: *aCIPType*

MODE aMode



PROCESS aProcessC

A process is a state machine which can be extended by variables, record types for inputs and outputs and operations executable in state and mode transitions. A process can have several modes. A mode defines a possible behaviour of the process and is graphically specified by means of a state transition diagram. The moderator of a process is a further transition structure acting on the process modes defining the possible changes of behaviour.

EXTENSIONS TIMER, CHAIN

Timer extension (syntactic sugar)
Every process can be extended by a local timer which is a TICK driven virtual process interacting with its owner. A timer can be set, reset or stopped in state transitions by means of emitted virtual SET TIMER and STOP TIMER outpulses. An expired timer triggers the timed process by the virtual TIMEUP pulse. An occurring TIMEUP inpulse activates the cascade of the timed process.

Chain extension (syntactic sugar)
The CHAIN extension is used to specify self triggering.
Every process can be extended by a virtual chain responder. The responder is activated in state transitions by means of an emitted virtual SET CHAIN outpulse. The responder comes back with a virtual CHAIN inpulse as soon as there are no external events pending. An occurring CHAIN inpulse activates the cascade of the chained process.

VARIABLES aVariable1: INTEGER, aVariable2: *aUserType*, aVariable3: *aCIPType*

Variables extend the context of a process. The type associated with a variable is either elementary or it is a CIP type or a user type (typedef).

EVENTS anEvent: *aCIPType*

Mode input.
Events occur in the physical environment. A transmitted event must be awaited in the current state of the process, otherwise a context error is generated (exception handling).

INPULSES anInpulse: *aCIPType*

Mode input.
Inpulses are generated by outpulses emitted by other processes of the cluster. Not awaited inpulses are lost.

COMMANDS aCommand: *aCIPType*

Moderator input.
Commands are generated by outpulses emitted by other processes of the cluster. Not awaited commands are lost.

INPORT anInport: *aPortType*

Mode input.
Inports are connected with datastreams (asynchronous communication).
All processes of the SteamBoilerController system are externally driven state machines with at most one inport. (A general CIP specification can also have polling processes with several inports.) The associated port type defines the incoming messages. A transmitted message must be awaited in the current state of the process, otherwise a context error is generated (exception handling).

TRIGGERS aTrigger

Moderator output, mode input.
Triggers are generated by the moderator of the process to activate a first state transition in the new mode immediately after the mode transition.

GATE1 anOutpulse: *aCIPType*

Mode output.
Outpulses are transmitted to other processes of the cluster. They are received either as inpulse in the process modes or as commands of the moderator.
The outpulse interface of a process consists of an ordered set of gates denoted as GATE1, GATE2, etc. Every outpulse of a process must be associated with one of its gates. Gates allow to specify branchings in the control flow of a cluster (see cascade).

GATE2 anOutpulse1: *aCIPType*, anOutpulse2

ACTIONS anAction: *aCIPType*

Mode output.
Actions are transmitted to the physical environment.

OUTPORT anOutport: *aPortType*

Mode output.
Outports are connected with datastreams (asynchronous communication).
A process can have several outports. The associated port type defines the outgoing messages.

OPERATION anOperation
```
{ SELF.aVariable1 = 5; }
```
A process operation is specified as ANSI C compound statement and can be allocated to state and mode transitions of the process. A variable of the process called aVariable1 is referenced as `SELF.aVariable1`. Fields of inputs and outputs are referenced as `IN.aField1`, `OUTPULSE.aField1`, `ACTION.aField1` and `OUTMSG.aField1` respectively, where aField1 is the name of a field of the associated CIP type.

OPERATION inputDataToOutpulse
```
{ OUTPULSE = IN; }  /*copy input record to output record */
```

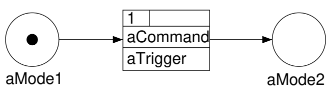
OPERATION storedDataToAction
```
{ ACTION.aField1 = SELF.aVariable1 }
```

OPERATION storeInputData
```
{ SELF.aVariable3.afield1 = IN.aField1; }
```

OPERATION useAnInquiry
```
{ SELF.aVariable1 = aProcessB.anInquiry(); }
```

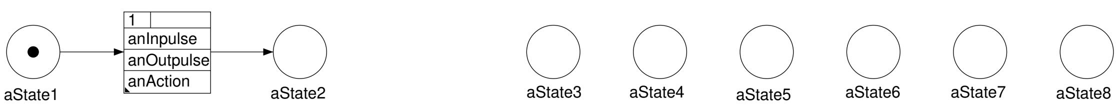
DELAY aDelay
```
SELF.aVariable1 * anIntConstant
```
Delays are specified as ANSI C expressions (number of TICKS). The duration between two TICKS is defined in the implementation of a CIP machine. Delays are used when a timer is set.

MODERATOR

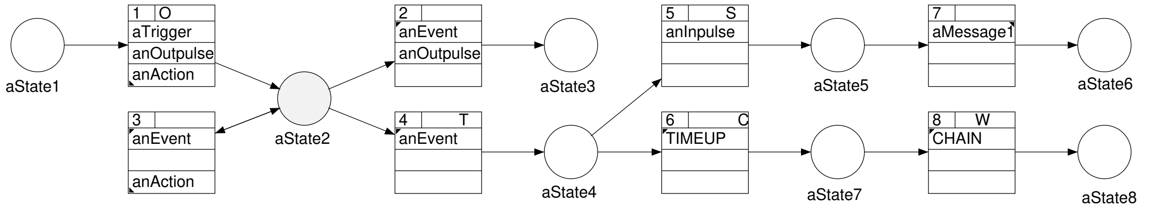


Mode transition diagram.
A moderator is a transition structure on the modes of a process. The inputs of a moderator are called commands. Commands are generated by outpulses of other processes of the same cluster. A mode transition fires when an awaited command occurs. The current state of a process is not changed in a mode transition. However, a moderator can generate triggers which activate a state transition in the new mode immediately after the mode transition. The initial mode is marked by a token in the mode transition diagram.

MODE aMode1



MODE aMode2



State transition diagram.
The behaviour of a mode is specified by a transition structure on the states of a process. A process can have several alternative modes. A state transition fires when an awaited input occurs. The input of a mode consists in general of events, messages, inpulses and triggers. The general output of a state transition consists of an outpulse, an action and of a message for every outport. The initial state is marked by a token in the state transition diagram of the initial mode. aState2 - shown in grey - is a branching state (several leaving transitions with identical input).

Legend for corner marks of transition boxes:

Top left: input is an event *Top right: input is a message*
*Bottom left: output is an ac*tion

Info line of transition boxes (top line):

O - OPERATIONS are allocated *C - SET CHAIN is allocated*
T - SET TIMER is allocated *W - WRITES are specified (messages)*
S - STOP TIMER is allocated

BRANCHING STATE *aState2* EVENT *anEvent: aCIPType*
OPERATIONS *storeInputData*
POSTSTATE *aState2* IF
`IN.aField1 == 5`
POSTSTATE *aState3* IF
`(aProcessB.anInquiry() == 1) && (STATUS.aVariable1 == 3)`
ELSE POSTSTATE *aState4*

A branching consists of a set of transitions starting in the same state triggered with the same input. States with one or several branchings are shown in grey. In order to resolve the non-determinism of a branching mutually exclusive conditions have to be associated with all branches. One branch can be defined as ELSE case.

Operations allocated to a branching are executed before the evaluation of the branching conditions. Such operations perform typically calculations used in several conditions of the branching.

TRANSITION 1
OPERATIONS *anOperation, inputDataToOutpulse, storedDataToAction, storeInputData*

TRANSITION 4
SET TIMER aDelay

TRANSITION 5
STOP TIMER

TRANSITION 6
SET CHAIN

TRANSITION 8
OUTPORT *anOutport* MESSAGE *aMessage2*

All transitions with specifications not fully visible in the transition diagram are listed:
- allocated operations, marked by O in the transition box
- specification of SET TIMER, STOP TIMER and SET CHAIN marked by S, T or C in the transition box
- written messages, marked by W in the transition box

PROCESS aProcessD

EVENTS anEvent

INPULSES anInpulse

GATE1 anOutpulse1, anOutpulse2

3. CIP Problem Domain of the Steam Boiler Controller

The software of the complete steam boiler control program developed with CIP consists of the code of the reactive layer generated from the CIP specification presented in the following chapters and of the code of the transmission layer which has to be developed using appropriate implementation techniques. The implementation of the transmission cycles described in the original problem description for instance is part of the development of the I/O-system of the transmission layer and is not treated here.

Abstraction from transmission cycles

The original description of the steam boiler problem is based on repetitive transmission cycles of 5 seconds, in which information is passed between the physical units and the control program. We consider such a cycle as a special property of the implementation of the transmission layer. In a CIP Specification one abstracts from such properties, describing the functional behaviour of the control program as a set of cooperating state machines.
In the real environment we identify only two physical processes with continuous states, namely the water level q and the steam flow v, which will lead to two processes of the reactive layer driven by repetitive sampling events. All other processes of the plant and of the operator desk have discrete states and will interact with the control system by means of asynchronous discrete events and actions.

Abstraction from transmission errors

The program must react on transmission errors generating the message EMERGENCY STOP for the physical environment. The detection of transmission errors is part of the tasks of the processes of the transmission layer. A CIP specification uses abstract services of the transmission layer which is supposed to be reliable. In the presented solution the transmission layer is supposed to generate for an occurring transmission error the event COM_ERR. The process Stopper reacts on this event simply by generating the action EM_STOP for the environment.

An improvement to the original steam boiler description: Fail save pumps

In an early CIP Specification of the steam boiler controller we followed the original requirement description as close as possible. However, the simulation of this specification raised the following problems:

a) It is very difficult to interpret an invalid occurrence of an event of a pump flow sensor. The original problem description states that "if the program knows from elsewhere that the pump is working correctly it is a pump controller failure, otherwise it is a pump failure". This interpretation gets especially unclear when other pumps are already broken.

b) Rescue mode is allowed for broken pumps. In such a case, the calculated values of the water level will fastly reach the critical limit M1 or M2 respectively because the recursive calculation of the water level is partly based on the maximal and minimal pump flow values P and 0 (monotone increase of uncertainty).

We solve these problems supposing *fail save pumps*:

If the physical environment is informed by the program that a pump or a pump control device has failed, the environment *disables* the corresponding pump until the corresponding reparation is completed and reported to the control system. The water flow from a disabled pump is guaranteed to be zero.

Improvement to the original problem description

All pumps are fail save in the sense mentioned above. If a pump or a pump control device fails the system is controlled like a system with N-1 pumps where N is the number of enabled pumps before the failure. If more then CRITICAL_PUMPNUMBER pumps must be disabled the system has to be stopped immediately.

4. Events, Actions and Modelprocesses of the Steam Boiler Controller

4.1. EVENT, ACTION and MESSAGE Lists

The development of a CIP specification starts with the definition of its input and output domain described as collections of events, actions and messages. This first development step already determines the level of abstraction of the reactive layer of the reactive system.

The following event, action and message lists contain implementation remarks related to the cyclic implementation proposed in the original problem description. Most acknowledge messages described in the original problem description do not appear in the event or action list because they concern the transmission layer only.

EVENT List

EVENTS	Comment
STOP	Stop message sent by the operator. Implementation: Received as message STOP
SB_WAIT	Start trigger for the initialization phase. Implementation: Received as message STEAM_BOILER_WAITING
PU_READY	Ready trigger for the working phase. Implementation: Received as message PHYSICAL_UNITS_READY
openACK	A pump has successfully opened. Implementation: Extracted from message PUMP_STATE(n,b)
closeACK	A pump has sucessfully closed. Implementation: Extracted from message PUMP_STATE(n,b)
P_FLOW	A pump flow sensor detects that the water starts to flow. Implementation: Extracted from message PUMP_CONTROL_STATE(n,b)
P_IDLE	A pump flow sensor detects that the water stops to flow. Implementation: Extracted from message PUMP_CONTROL_STATE(n,b)
Level_q	Sampling event of the water level sensor: level q with range [0, C]. Implementation: Received as message LEVEL(v)
Flow_v	Sampling event of the steam flow sensor: flow v with range [0, W]. Implementation: Received as message STEAM(v)
PumpREP	A failed pump has been repaired. Implementation: Received as message PUMP_REPAIRED(n)
PCtrl_REP	A failed pump control device has been repaired. Implementation: Received as message PUMP_CONTROL_REPAIRED(n)
WaterREP	The water level sensor has been repaired. Implementation: Received as message LEVEL_REPAIRED
SteamREP	The steam flow sensor has been repaired. Implementation: Received as message STEAM_REPAIRED
COM_ERR	A communication error has occurred. Implementation: Generated by the transmission layer

ACTION List

ACTION	Comment
EM_STOP	The physical environment must immediately stop the system. Implementation: Sent as message MODE(EMERGENCY STOP)
PR_READY	The initialization phase has successfully completed. Implementation: Sent as message PROGRAM_READY
OPEN_V	The valve must be opened. Implementation: Sent as message VALVE
CLOSE_V	The valve must be closed. Implementation: Sent as message VALVE
OPEN_P	The pump must be opened. Implementation: Sent as message OPEN_PUMP(n)
CLOSE_P	The pump must be closed. Implementation: Sent as message CLOSE_PUMP(n)
PumpFAIL	A failure of a pump has been detected. Implementation: Sent as message PUMP_FAILURE_DETECTION(n)
PCtrl_FAIL	A failure of a pump control device has been detected. Implementation: Sent as message PUMP_CONTROL_FAILURE_DETECTION(n)
WaterFAIL	A failure of the water level sensor has been detected. Implementation: Sent as message LEVEL_FAILURE_DETECTION
SteamFAIL	A failure of the steam flow sensor has been detected. Implementation: Sent as message STEAM_FAILURE_DETECTION

MESSAGE List

The listed messages inform the operator about the actual mode of operation of the control system. They have no effect on the behaviour of the physical environment and are functionally determined by the history of failure actions and reparation events.

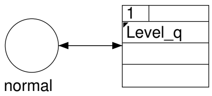
MESSAGE	Comment
Normal	Normal control operation. No pending failures. Implementation: Sent as message MODE(NORMAL)
Degraded	Normal control operation despite of occurred failures. Implementation: Sent as message MODE(DEGRADED)
Rescue	Control is based on calculated water level values. Implementation: Sent as message MODE(RESCUE)

4.2. Model Processes

Model processes are defined as event driven and action generating state machines. The transition structure of a model process reflects the normal behaviour of the corresponding object of the physical environment. It defines in fact all valid time orderings of occurring events and generated actions of a process (sequential protocols).

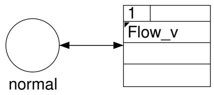
The result of the modelling phase is a set of independent incomplete state machines forming the basis for the development of the functional behaviour of the control system in a second phase (Section 5).

PROCESS WaterLevel



EVENTS Level_q

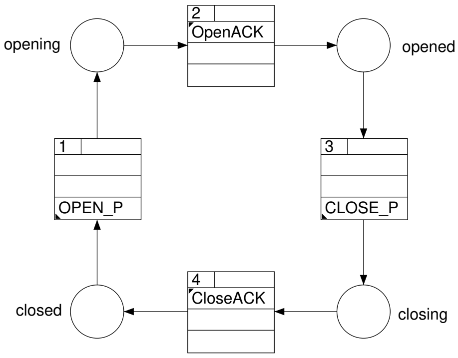
PROCESS SteamFlow



EVENTS Flow_v

The sequential structure of processes driven by repetitive sampling events is trivial. Information is transmitted by the data of the sampling events.

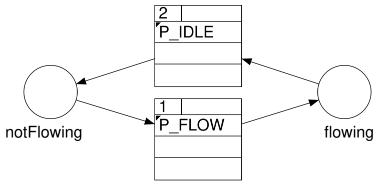
PROCESS Pump



EVENTS CloseACK, OpenACK
ACTIONS CLOSE_P, OPEN_P

Interface process to a pump.

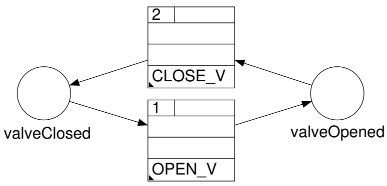
PROCESS Flow



EVENTS P_FLOW, P_IDLE

Interface process to a pump control device.

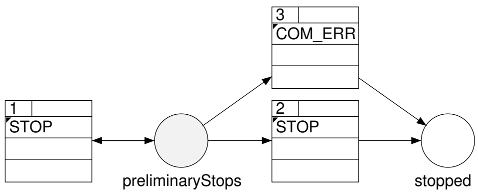
PROCESS Valve



ACTIONS CLOSE_V, OPEN_V

Interface process to the boiler valve.

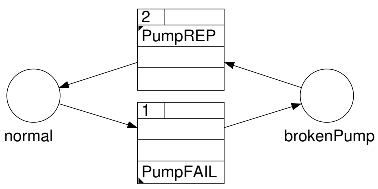
PROCESS Stopper



EVENTS COM_ERR, STOP

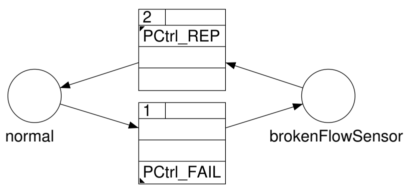
The boiler controller is stopped when a transmission error occurs or when the operator executes three stop commands.

PROCESS PumpMaster



EVENTS PumpREP
ACTIONS PumpFAIL

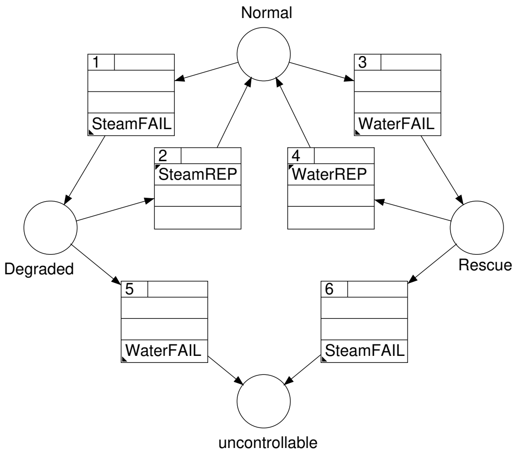
PROCESS FlowMaster



EVENTS PCtrl_REP
ACTIONS PCtrl_FAIL

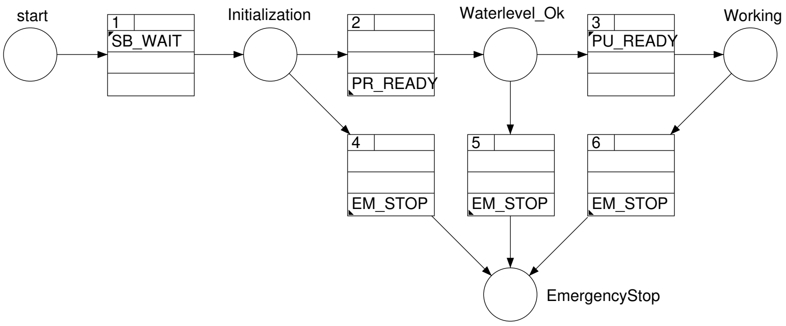
PumpMaster and FlowMaster will be extended to failure managers for the pumps and the pump control devices.

PROCESS BoilerMaster



EVENTS SteamREP, WaterREP
ACTIONS SteamFAIL, WaterFAIL
The current state of BoilerMaster will determine the actual mode of operation of the steam boiler controller.

PROCESS SystemLife



EVENTS PU_READY, SB_WAIT
ACTIONS EM_STOP, PR_READY
SystemLife will control the life phases of the steam boiler controller.

5. CIP Specification of the Steam Boiler Controller

We explain our solution of the steam boiler problem within the comments of a system report generated from a specification developed with *CIP Tool®* [3]. The report gives a formally complete description of the specified steam boiler controller. In order to get a more compact presentation for this paper we edited the layout of the generated report by hand.

The general structure of a CIP system report, the meaning of the used report fonts and the meaning of the particular objects of a CIP specification are explained by means of the generated report of the dummy system *ReportHelp* of Section 3.

The specification has been developed in a compositional constructive way, starting with the model processes defined in the previous section. As further components function processes have been introduced and through specification of interaction and communication the functional reactive behaviour of the control system has been defined.

SYSTEM SteamBoilerController

The specified system satisfies all requirements of the original steam boiler problem description. However, in order to be able to develop a satisfactory failure management for the pumps the original description has been improved supposing fail save pumps.

Two main problems have been encountered during the specification:
a) On one hand an occurring invalid event of a pump flow sensor must be interpreted as pump failure when the sensed water level is incompatible with the calculated values, otherwise it is taken as a flow sensor failure. On the other hand an incompatible behaviour of the water level is interpreted as a water level sensor failure. The problem is solved as follows: If the last water level value has been compatible, the interpretation of an invalid flow sensor event is delayed in order to allow the program to detect an incompatible water level value (pump error propagation delay). If after the delay the water level has got incompatible the pump is considered as broken and disabled immediately, thus future water level values are expected to be correct again. However, if a detected incompatible water level remains incompatible in the following sampling cycle the water level sensor is considered as broken.

b) Water level values calculated for the next sampling cycle can be incorrect if one or several pumps have been disabled due to broken pumps and broken pump flow sensors between two sampling cycles. The problem is solved by correcting previously calculated water level values appropriately.

The original problem description defines five modes of operation. We consider only Normal, Degraded and Rescue as modes of operation. Initialization and Emergency stop are taken as phases of the systems life.

For the estimation of the water level and the steam flow we apply the algorithm proposed in the document "steam boiler additional information" accompanying the original problem description [2]. We adopt the proposed notation for calculated and adjusted values of the water level, the steam flow and the pump flow. We call them q-values, v-values and p-values.

CLUSTER List
BoilerControl, *PumpControl*

BoilerControl is the main controller of the SteamBoilerController system supervising the water level and the steam flow of the boiler. The water level is controlled using the services of the PumpControl cluster which is designed as reusable software component. The failure management concerning the water and the steam sensor is part of the tasks of the BoilerControl cluster. Pump and pump control device failures are handled by the PumpControl cluster.

INTEGER CONSTANTS
CRITICAL_PUMPNUMBER: 3, PUMPNUMBER: 4, TICKS_PER_SECOND: 10

FLOAT CONSTANTS
C: 20.0, D: 4.0, M1: 1.0, M2: 19.0, N1: 6.0, N2: 14.0, P: 2.0, U1: 1.1, U2: 1.1, W: 3.0,
CT: 0.5,
FLOW_DECREASE_DELAY: 0.2, FLOW_INCREASE_DELAY: 0.9,
PUMP_ACK_DELAY: 0.1, PUMPERROR_PROPAGATION_DELAY: 0.9,
PUMPFLOW_INCREASE: 2.0, PUMP_VARIATION: 0.2, DRAIN_VARIATION: 0.2

For constants already appearing in the original problem description we use the abbreviated symbols proposed there. CT denotes the time between two sampling cycles. Additionally introduced constants have descriptive names.

CIP TYPE qAdjusted
FIELDS qa1: FLOAT, qa2: FLOAT

CIP TYPE qRaw_qCalculated
FIELDS q: FLOAT, qc1: FLOAT, qc2: FLOAT

CIP TYPE qRaw_vRaw
FIELDS q: FLOAT, v: FLOAT

CIP TYPE vRaw
FIELDS v: FLOAT

PORT TYPE doubtMessages
MESSAGES InDoubt, NoDoubt

PORT TYPE modeMessages
MESSAGES Degraded, Normal, Rescue

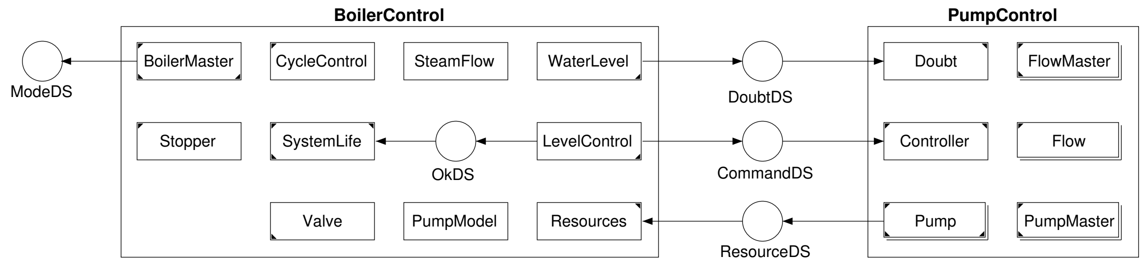
PORT TYPE pumpMessages
MESSAGES Closing, Opening

PORT TYPE resourceMessages
MESSAGES Disabled, Enabled

PORT TYPE OkMessage
MESSAGES Level_Ok

INDEX TYPE PumpRow [*PUMPNUMBER*]

COMMUNICATION NET SteamBoilerNet



The SteamBoilerNet defines the asynchronous communication between the BoilerControl and the PumpControl cluster. The net shows the two clusters with all its processes. The pulse flow between the processes within a cluster is defined in the cluster interaction nets and is hidden here.

Meaning of the datastreams:

DoubtDS informs the PumpControl cluster whether the correct working of the pumps is doubted.

CommandDS transmits opening and closing messages for the pumps to the PumpControl cluster.

ResourcesDS transmits a message whenever a pump is enabled or disabled.

ModeDS is an external datastream informing the operator about the actual mode of operation.

OkDS is an internal datastream of the BoilerControl cluster used to signal SystemLife the termination of the initialization phase.

DATASTREAM CommandDS [8]
READER *Controller.cmd_in* WRITER *LevelControl.cmd_out*

DATASTREAM DoubtDS [4]
READER *Doubt.dbt_in* WRITER *WaterLevel.dbt_out*

DATASTREAM ModeDS [1]
WRITER *BoilerMaster.mdOut*

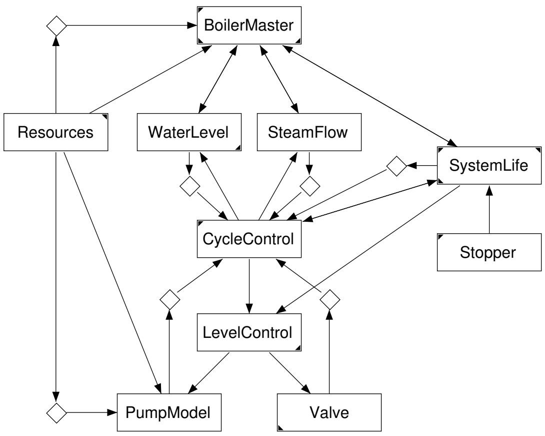
DATASTREAM OkDS [1]
READER *SystemLife.ok_in* WRITER *LevelControl.ok_out*

DATASTREAM ResourceDS [4]
READER *Resources.res_in* WRITER *Pump.res_out*

CLUSTER BoilerControl

Model processes: WaterLevel, SteamFlow, Valve, SystemLife, BoilerMaster, Stopper
Function processes: CycleControl, LevelControl, PumpModel, Resources

INTERACTION NET



SystemLife controls the life of the steam boiler system informing several processes about the system start and the readiness of the physical environment. Several processes can demand to stop the system.

Stopper can cause a system stop.

BoilerMaster is responsible for the failure management concerning WaterLevel and SteamFlow. The process determines the actual mode of operation which depends also on the number of enabled pumps stored by Resources.

CycleControl receives samples of the water level and the steam flow as singular sampling event. The process is responsible for the calculation and adjustment of the water level values and for triggering those processes which have to perform cluster functions related to the sampling cycle. The calculation of the water level values depends on the states and the variables of several processes (inspection). CycleControl supervises also the critical water level limits and the steam flow value in the initialization phase (stop to SystemLife).

WaterLevel decides about the reliability of the water sensor. A detected failure of the water level sensor is signalled to BoilerMaster which confirms when the sensor has been repaired. The process writes also doubt messages to the PumpControl cluster when the correct working of the pumps is doubted.

SteamFlow estimates and adjusts the steam flow values. A detected failure of the steam flow sensor is signalled to BoilerMaster which in turn reports when the sensor has been repaired.

LevelControl writes opening and closing commands to the PumpControl cluster depending on the adjusted water level values. PumpModel is triggered in every cycle for updating the modelled pump flow values. In the initialization phase LevelControl controls also the boiler valve.

PumpModel calculates approximate pump flow values for opening and closing pumps respectively. The total adjusted pump flow is calculated using information about the total pump activity available from Resources (inspection). PumpModel counts also the number of disabled pumps between two sampling cycles. This number is used by CycleControl to correct calculated water level values.

Valve is triggered by LevelControl to open and close the valve.

Resources counts the total number of enabled pumps. Changes to the number of enabled pumps are reported to BoilerMaster and PumpModel.

PULSE TRANSLATIONS

SENDER *BoilerMaster*
- *steamRep -> SteamFlow.steamRep*
- *stop -> SystemLife.stop*
- *waterRep -> WaterLevel.waterRep*

SENDER *CycleControl*
- *control -> LevelControl.control*
- *flow_v -> SteamFlow.flow_v*
- *level_q -> WaterLevel.level_q*
- *stop -> SystemLife.stop*

SENDER *LevelControl*
- *closed_P -> PumpModel.closed_P*
- *close_V -> Valve.close_V*
- *opened_P -> PumpModel.opened_P*
- *open_V -> Valve.open_V*
- *p_adjust -> PumpModel.p_adjust*

SENDER *Resources*
- *allEnabled -> BoilerMaster.allEnabled*
- *critDisable -> BoilerMaster.critDisable*
- *disabled -> BoilerMaster.disabled, PumpModel.disabled*

SENDER *SteamFlow*
- *steamFail -> BoilerMaster.steamFail*

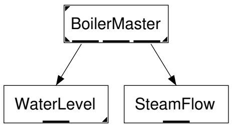
SENDER *Stopper*
- *stop -> SystemLife.stop*

SENDER *SystemLife*
- *ready -> BoilerMaster.ready, LevelControl.ready*
- *start -> CycleControl.start*

SENDER *WaterLevel*
- *waterFail -> BoilerMaster.waterFail*

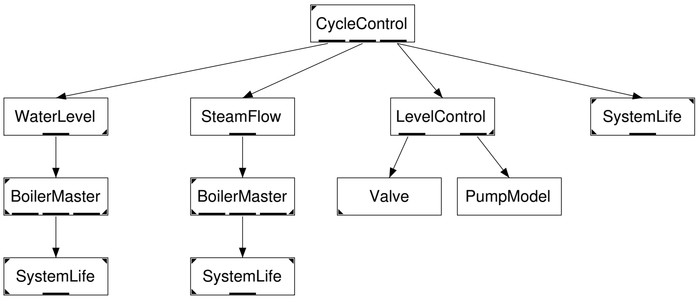
For expository reasons we used the same name for a received inpulse as for the sent outpulse which forced us to introduce some extra transitions in some transition structures.

CASCADE *BoilerMaster*



Reparation events concerning the water level and the steam flow sensor are signalled to WaterLevel and SteamFlow.

CASCADE *CycleControl*



The CycleControl cascade defines the control flow for the pulse transmissions activated in the four state transitions of CycleControl sequentially triggered when the sampling event occurs:

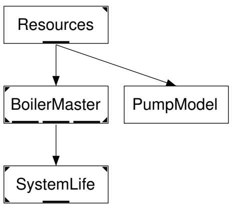
The pulse *level_q* of the left gate of CycleControl transmits the raw and the calculated q-values to WaterLevel. WaterLevel verifies the actual q-value. A detected failure of the water sensor is signalled to BoilerMaster which can cause a system stop (SystemLife).

The pulse *flow_v* of the middle gate of CycleControl transmits the sampled v-value to SteamFlow. A detected failure of the steam flow sensor is signalled BoilerMaster which can cause a system stop (SystemLife).

The pulse *control* of the right gate of CycleControl transmits the adjusted q-values to LevelControl which is responsible for opening and closing the valve and the pumps. LevelControl triggers also PumpModel for updating.

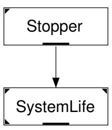
The pulse *stop* of the right gate of CycleControl is used to alarm SystemLife when the calculation of the q-values shows that the critical limits can be reached.

CASCADE *Resources*



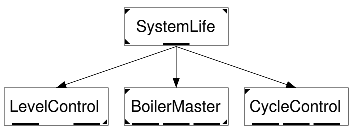
Resources receives messages from the PumpControl cluster whenever a pump is disabled. BoilerMaster and PumpModel are informed about enabled and disabled pumps. When the number of enabled pumps gets below the critical pump number the alerted BoilerMaster causes a system stop (SystemLife).

CASCADE *Stopper*



Three stop events or a communication error lead to a system stop (SystemLife).

CASCADE *SystemLife*



The system start and the readiness of the physical units have to be signalled to several processes.

PROCESS BoilerMaster

BoilerMaster has already been introduced as model process. The elaborated process is responsible for the failure management concerning the water level and the steam flow sensor and determines the actual mode of operation of the steam boiler controller (modes of operations correspond to states of BoilerMaster). Changes of the actual mode of operation are reported to the operator desk by messages written through outport md_out. BoilerMaster is informed by WaterLevel and SteamFlow about occurring failures, reporting them in turn when a broken sensor has been repaired. BoilerMaster is also informed about disabled pumps in order to determine the actual mode of operation and to cause a system stop when to much pumps are disabled.

The failure management in the initialization phase differs from the one in the working phase which is reflected by the to two modes initialization and working of the process.

EVENTS SteamREP, WaterREP

INPULSES allEnabled, critDisable, disabled, steamFail, waterFail

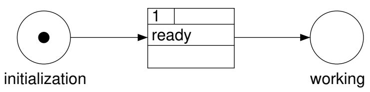
COMMANDS ready

GATE1 waterRep
GATE2 stop
GATE3 steamRep

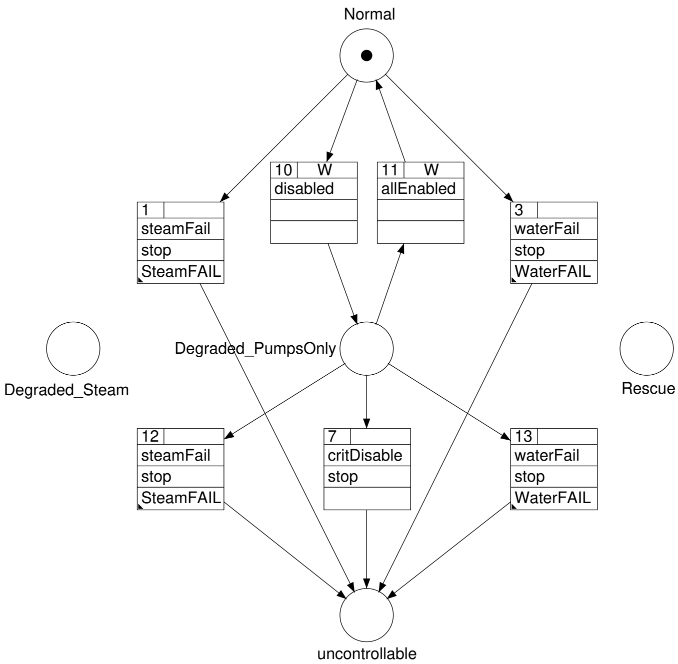
ACTIONS SteamFAIL, WaterFAIL

OUTPORT mdOut: *modeMessages*

MODERATOR



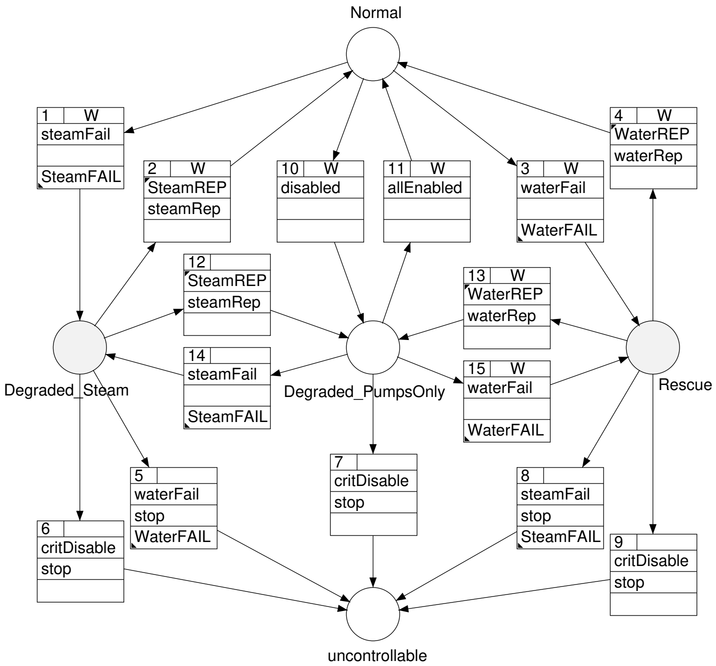
MODE initialization



TRANSITION 10
OUTPORT *mdOut* MESSAGE *Degraded*

TRANSITION 11
OUTPORT *mdOut* MESSAGE *Normal*

MODE working



BRANCHING STATE *Degraded_Steam* EVENT *SteamREP*
POSTSTATE *Normal* IF
`Resources.allPumpsEnabled()`
ELSE POSTSTATE *Degraded_PumpsOnly*

BRANCHING STATE *Rescue* EVENT *WaterREP*
POSTSTATE *Normal* IF
`Resources.allPumpsEnabled()`
ELSE POSTSTATE *Degraded_PumpsOnly*

TRANSITIONS 1, 10, 13
OUTPORT *mdOut* MESSAGE *Degraded*

TRANSITIONS 2, 4, 11
OUTPORT *mdOut* MESSAGE *Normal*

TRANSITIONS 3, 15
OUTPORT *mdOut* MESSAGE *Rescue*

PROCESS CycleControl

The process is activated periodically by the sampling event Sample carrying the sensed values of the water level and the steam flow. The process performs the calculations and adjustments concerning the water level and triggers all processes which perform functions concerning the cyclic control algorithm of the BoilerControl cluster. The process supervises also the critical limits of the sensed values. The activities of CycleControl are already described in detail in the comment of the cascade of CycleControl.

A problem:

Pump and pump control failures occur asynchronously to the sampling cycle for the water level and the steam flow. Thus the previously calculated q-values may be incorrect. The problem is solved by correcting the calculated q-values using the corrected total pump flow value available from PumpModel.

EXTENSIONS CHAIN

VARIABLES q: FLOAT, qa1: FLOAT, qa2: FLOAT, qc1: FLOAT, qc2: FLOAT, v: FLOAT

EVENTS SAMPLE: *qRaw_vRaw*

INPULSES start

GATE1 level: *qRaw_qCalculated*
GATE2 flow_v: *vRaw*
GATE3 control: *qAdjusted*, stop

OPERATION adjust_q
```
{
      if (WaterLevel.broken()){
            SELF.qa1 = SELF.qc1; SELF.qa2 = SELF.qc2;
      }
      else{
            SELF.qa1 = SELF.q; SELF.qa2 = SELF.q;
      }
}
```

OPERATION calculate_q
```
{
  SELF.qc1 =
       SELF.qa1 - SteamFlow.va2()*CT - 0.5*U1*CT*CT + PumpModel.pa1()*CT - Valve.da2()*CT;
  SELF.qc2 =
       SELF.qa2 - SteamFlow.va1()*CT + 0.5*U2*CT*CT + PumpModel.pa2()*CT - Valve.da1()*CT;
}
```

OPERATION correct_qc
```
{ SELF.qc1 = SELF.qc1 - PumpModel.pa1_correction()*CT; }
```

OPERATION initialize_qc
```
{ SELF.qc1 = 0; SELF.qc2 = C; }
```

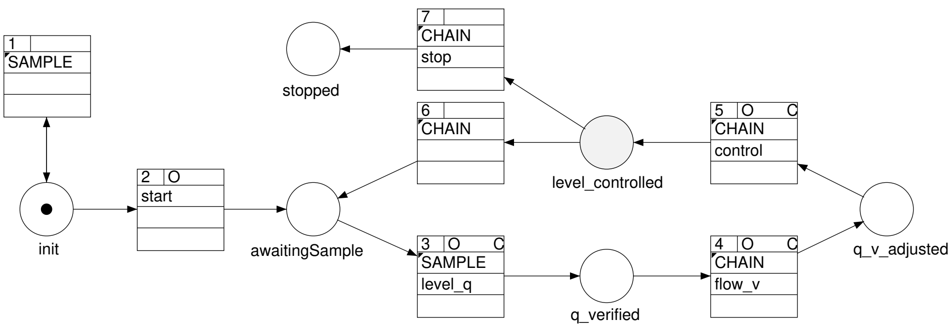
OPERATION qa_toOutpulse
```
{ OUTPULSE.qa1 = SELF.qa1; OUTPULSE.qa2 = SELF.qa2; }
```

OPERATION qc_toOutpulse
```
{ OUTPULSE.q = SELF.q; OUTPULSE.qc1 = SELF.qc1; OUTPULSE.qc2 = SELF.qc2; }
```

OPERATION store_sampling
```
{ SELF.q = IN.q; SELF.v = IN.v; }
```

OPERATION v_toOutpulse
```
{ OUTPULSE.v = SELF.v; }
```

MODE unique



BRANCHING STATE *level_controlled* INPULSE *CHAIN*
    OPERATIONS *calculate_q*
    POSTSTATE *stopped* IF
```
(SELF.qc1 < M1 || SELF.qc2 > M2) ||
(SystemLife.notReady() && (STATUS.v != 0 || WaterLevel.broken()))
```
    ELSE POSTSTATE *awaitingSample*

TRANSITION 2
    OPERATIONS *initialize_qc*

TRANSITION 3
    SET CHAIN
    OPERATIONS *store_sampling, correct_qc, qc_toOutpulse*

TRANSITION 4
    SET CHAIN
    OPERATIONS *adjust_q, v_toOutpulse*

TRANSITION 5
    SET CHAIN
    OPERATIONS *qa_toOutpulse*

PROCESS LevelControl

This process, fully triggered by CycleControl, is responsible for opening and closing the valve and the pumps. The decision depends on the adjusted water level transmitted by CycleControl. LevelControl also triggers PumpModel for updating.

The process has two modes corresponding to the controlling in the initialization and in the working phase. When the initialization phase is complete SystemLife is informed by means of a written Level_Ok message.

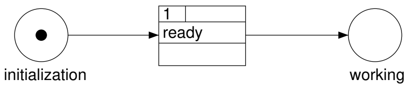
INPULSES control: *qAdjusted*

COMMANDS ready

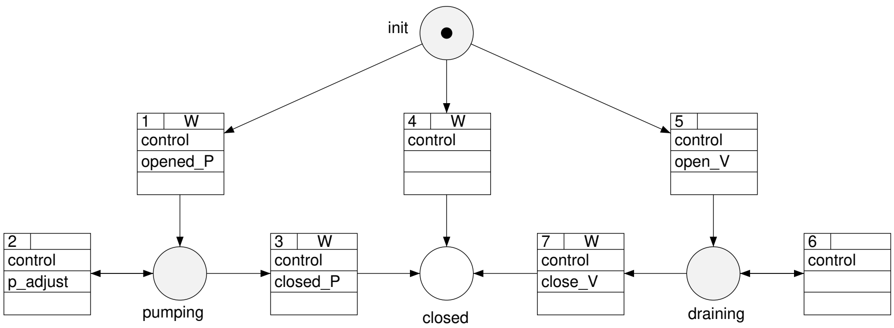
GATE1 close_V, open_V
GATE2 closed_P, opened_P, p_adjust

OUTPORT cmd_out: *pumpMessages*
OUTPORT ok_out: *OkMessage*

MODERATOR



MODE initialization



BRANCHING STATE *init* INPULSE *control: qAdjusted*
    POSTSTATE *draining* IF
    `IN.qa2 >= N2`
    POSTSTATE *pumping* IF
    `IN.qa1 <= N1`
    ELSE POSTSTATE *closed*

BRANCHING STATE *draining* INPULSE *control: qAdjusted*
    POSTSTATE *closed* IF
    `IN.qa2 < N2`
    ELSE POSTSTATE *draining*

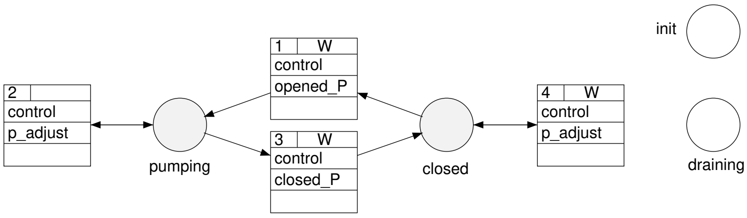
BRANCHING STATE *pumping* INPULSE *control: qAdjusted*
    POSTSTATE *closed* IF
    `IN.qa1 > N1`
    ELSE POSTSTATE *pumping*

TRANSITION 1
    OUTPORT *cmd_out* MESSAGE *Opening*

TRANSITION 3
    OUTPORT *cmd_out* MESSAGE *Closing*
    OUTPORT *ok_out* MESSAGE *Level_Ok*

TRANSITIONS 4, 7
    OUTPORT *ok_out* MESSAGE *Level_Ok*

MODE working



BRANCHING STATE *closed* INPULSE *control: qAdjusted*
POSTSTATE *pumping* IF
```
IN.qa1 < N1 && IN.qa2 < N2
```
ELSE POSTSTATE *closed*

BRANCHING STATE *pumping* INPULSE *control: qAdjusted*
POSTSTATE *closed* IF
```
IN.qa1 > N1 && IN.qa2 > N2
```
ELSE POSTSTATE *pumping*

TRANSITIONS
TRANSITION 1
OUTPORT *cmd_out* MESSAGE *Opening*

TRANSITION 3
OUTPORT *cmd_out* MESSAGE *Closing*

PROCESS PumpModel

The q-calculation (water level) during the opening period (5 sec.) of the pumps proposed in the "additional steam boiler information" is based on the minimal and maximal flow values 0 and P of the pumps. A simulation of an early specification has shown that this estimate is too rough in Rescue mode (monotone uncertainty increase). We refine the approximation using a pump model driven by the cycle rate of CycleControl. The minimal and maximal flow values of the opening pump are modelled by two linear step functions. The pumps are supposed to close within one sampling cycle. The model can easily be adjusted to the behaviour of the actual real pumps used in the system.

The current pump flow values of the process are used together with the number of enabled pumps available from Resources to calculate the total adjusted pump flow which is used in the q-calculation of CycleControl.

INQUIRY pa1
```
float pa1 (void)
{ return (STATUS.PumpsInUse * STATUS.pc1); }
```

INQUIRY pa1_correction
```
float pa1_correction (void)
{ return (STATUS.recentDisablings * STATUS.pc1); }
```

INQUIRY pa2
```
float pa2 (void)
{ return (STATUS.PumpsInUse * STATUS.pc2); }
```

VARIABLES pc1: FLOAT, pc2: FLOAT, PumpsInUse: INTEGER, recentDisablings: INTEGER

INPULSES closed_P, disabled, opened_P, p_adjust

OPERATION adjust_PumpsInUse
```
{ SELF.PumpsInUse = SELF.PumpsInUse - SELF.recentDisablings; }
```

OPERATION decrementFlow
```
{ SELF.pc2 = P; SELF.pc1 = 0; }
```

OPERATION incrementFlow
```
{ SELF.pc1 = SELF.pc2; SELF.pc2 = SELF.pc2 + PUMPFLOW_INCREASE*CT; }
```

OPERATION incr_recentDisablings
```
{ SELF.recentDisablings++; }
```

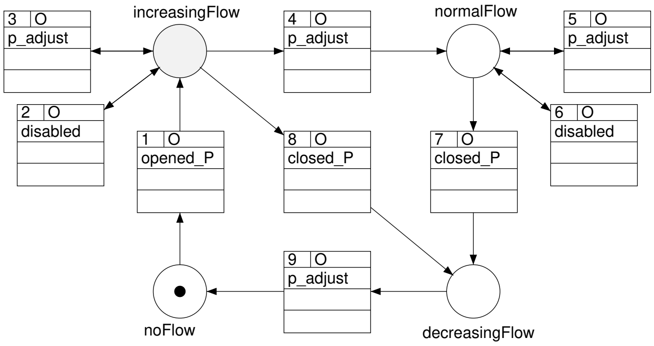
OPERATION reset_recentDisablings
```
{ SELF.recentDisablings = 0;}
```

OPERATION setFlowToP
```
{ SELF.pc2 = P + PUMP_VARIATION;  SELF.pc1 = P - PUMP_VARIATION; }
```

OPERATION setFlowToZero
```
{ SELF.pc2 = 0;  SELF.pc1 = 0; }
```

OPERATION set_PumpsInUse
```
{ SELF.PumpsInUse = Resources.N_enabled(); }
```

MODE unique



BRANCHING STATE *increasingFlow* INPULSE *p_adjust*
POSTSTATE *normalFlow* IF
`STATUS.pc2 >= P`
ELSE POSTSTATE *increasingFlow*

TRANSITION 1
OPERATIONS *set_PumpsInUse, reset_recentDisablings, incrementFlow*

TRANSITIONS 2, 6
OPERATIONS *incr_recentDisablings*

TRANSITION 3
OPERATIONS *adjust_PumpsInUse, reset_recentDisablings, incrementFlow*

TRANSITION 4
OPERATIONS *adjust_PumpsInUse, reset_recentDisablings, setFlowToP*

TRANSITION 5
OPERATIONS *adjust_PumpsInUse, reset_recentDisablings*

TRANSITIONS 7, 8
OPERATIONS *decrementFlow*

TRANSITION 9
OPERATIONS *setFlowToZero*

PROCESS Resources

The process is informed about enabled and disabled pumps by the PumpControl cluster. The number of disabled pumps is used by BoilerControl and PumpModel. Furthermore, every new disabling of a pump is signalled immediately to PumpModel which is responsible for the calculation of the pump flow corrections.

INQUIRY allPumpsEnabled
```
int allPumpsEnabled (void)
{ return STATUS.STATE == allPumpsEnabled; }
```

INQUIRY N_enabled
```
int N_enabled (void)
{ return (STATUS.N); }
```

VARIABLES N: INTEGER

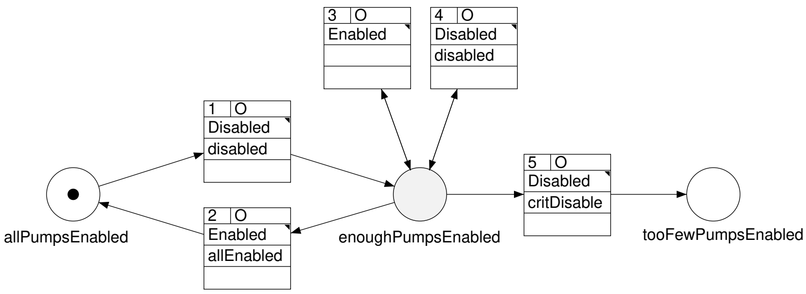
INPORT res_in: *resourceMessages*

GATE1 allEnabled, critDisable, disabled

OPERATION decr_N
```
{ SELF.N--; }
```

OPERATION incr_N
```
{ SELF.N++; }
```

MODE unique



BRANCHING STATE *enoughPumpsEnabled* MESSAGE *Disabled*
POSTSTATE *tooFewPumpsEnabled* IF
`STATUS.N == CRITICAL_PUMPNUMBER`
ELSE POSTSTATE *enoughPumpsEnabled*

BRANCHING STATE *enoughPumpsEnabled* MESSAGE *Enabled*
POSTSTATE *allPumpsEnabled* IF
`STATUS.N + 1 == PUMPNUMBER`
ELSE POSTSTATE *enoughPumpsEnabled*

TRANSITIONS 1, 4, 5
OPERATIONS *decr_N*

TRANSITIONS 2, 3
OPERATIONS *incr_N*

PROCESS SteamFlow

The sampling event *Flow_v* of the corresponding model process appears now as inpulse *flow_v* transmitted by CycleControl. The process performs the steam flow calculations used by CycleControl. SteamFlow is also responsible to detect failures of the steam flow sensor. Sensor failures are reported to BoilerMaster which in turn informs about reparations.

INQUIRY va1
```
float va1 (void)
{ return STATUS.va1; }
```

INQUIRY va2
```
float va2 (void)
{ return STATUS.va2; }
```

VARIABLES va1: FLOAT, va2: FLOAT, vc1: FLOAT, vc2: FLOAT

INPULSES flow_v: *vRaw*, steamRep

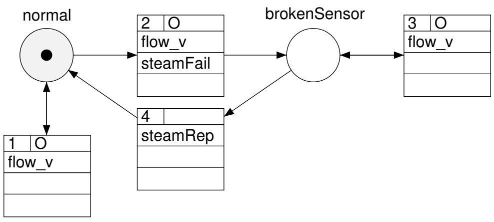
GATE1 steamFail

OPERATION adjustDegraded
```
{ SELF.va1 = SELF.vc1; SELF.va2 = SELF.vc2; }
```

OPERATION adjustNormal
```
{ SELF.va1 = IN.v; SELF.va2 = IN.v; }
```

OPERATION calculate
```
{
      SELF.vc1 = SELF.va1 - U2*CT; if (SELF.vc1 < 0) SELF.vc1 = 0;
      SELF.vc2 = SELF.va2 + U1*CT; if (SELF.vc2 > W) SELF.vc2 = W;
}
```

MODE unique



BRANCHING STATE *normal* INPULSE *flow_v: vRaw*
POSTSTATE *normal* IF
`IN.v >= STATUS.vc1 && IN.v <= STATUS.vc2`
ELSE POSTSTATE *brokenSensor*

TRANSITION 1
OPERATIONS *adjustNormal, calculate*

TRANSITIONS 2, 3
OPERATIONS *adjustDegraded, calculate*

PROCESS Stopper

When a transmission error occurs or when the operator executes three stop commands a stop outpulse for SystemLife will be generated.

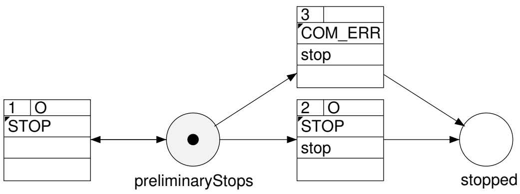
VARIABLES count: INTEGER

EVENTS COM_ERR, STOP

GATE1 stop

OPERATION incr
```
{ SELF.count++; }
```

MODE unique



BRANCHING STATE *preliminaryStops* EVENT *STOP*
POSTSTATE *preliminaryStops* IF
`STATUS.count < 2`
ELSE POSTSTATE *stopped*

TRANSITIONS 1,2
OPERATIONS *incr*

PROCESS SystemLife

The process controls the life of the SteamBoilerController system: initialization, working, emergency stop.

INQUIRY notReady

```
int notReady (void)
{ return (STATUS.STATE == Initialization || STATUS.STATE == Waterlevel_Ok); }
```

EVENTS PU_READY, SB_WAIT

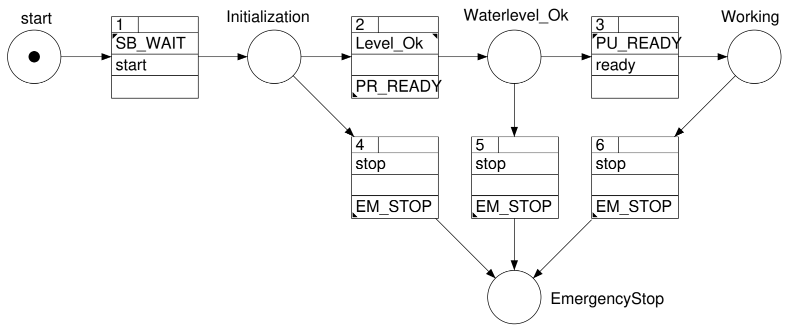
INPULSES stop

INPORT ok_in: *OkMessage*

GATE1 ready, start

ACTIONS EM_STOP, PR_READY

MODE unique



PROCESS Valve

The corresponding model process has been completed by inpulses. The process calculates the adjusted drain values used in the calculation of CycleControl.

INQUIRY da1

```
float da1 (void)
{
      if (STATUS.STATE == valveClosed) return 0;
      else return (D - DRAIN_VARIATION);
}
```

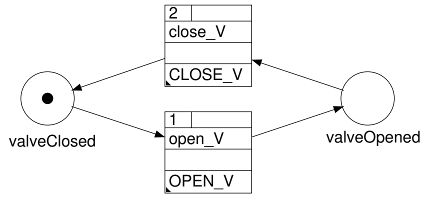
INQUIRY da2

```
float da2 (void)
{
      if (STATUS.STATE == valveClosed) return 0;
      else return (D + DRAIN_VARIATION);
}
```

INPULSES close_V, open_V

ACTIONS CLOSE_V, OPEN_V

MODE unique



PROCESS WaterLevel

The sampling event *Level_q* of the corresponding model process appears now as inpulse *level_q* transmitted by CycleControl. The process compares the raw value of the water level with the calculated values also transmitted by CycleControl.

When a first incorrect q-value occurs a doubt message is written to the PumpControl cluster. If the incorrect value cannot be the result of a broken pump, the error is taken as a water level sensor failure, otherwise the decision is postponed to the next cycle. If the reason for the incompatibility is a broken pump, the pump will be disabled and the q-value in the next cycle is expected to be correct. A second incorrect q-value is interpreted as water level sensor failure. Sensor failures are reported to BoilerMaster which in turn informs about reparations.

INQUIRY broken

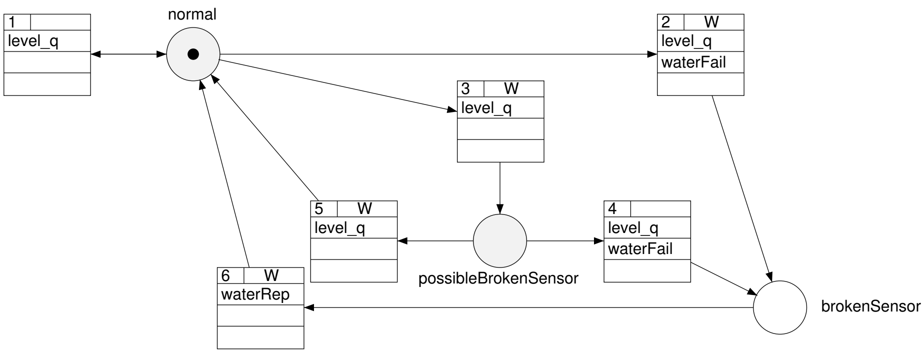
```
int broken (void)
{ return (STATUS.STATE != normal); }
```

INPULSES level_q: *qRaw_qCalculated*, waterRep

GATE1 waterFail

OUTPORT dbt_out: *doubtMessages*

MODE unique



BRANCHING STATE *normal* INPULSE *level_q: qRaw_qCalculated*

POSTSTATE *brokenSensor* IF

```
IN.q < 0 || IN.q > C
```

POSTSTATE *normal* IF

```
IN.q >= IN.qc1 && IN.q <= IN.qc2
```

ELSE POSTSTATE *possibleBrokenSensor*

BRANCHING STATE *possibleBrokenSensor* INPULSE *level_q: qRaw_qCalculated*

POSTSTATE *normal* IF

```
IN.q >= IN.qc1 && IN.q <= IN.qc2
```

ELSE POSTSTATE *brokenSensor*

TRANSITIONS 2, 3

OUTPORT *dbt_out* MESSAGE *InDoubt*

TRANSITIONS 5, 6

OUTPORT *dbt_out* MESSAGE *NoDoubt*

CLUSTER PumpControl

Model processes: Pump, PumpMaster, Flow, FlowMaster
Function processes: Controller, Doubt

This cluster contains processes with multiple instances: All model processes occur with the same multiplicity defined by index type PumpRow[PUMPNUMBER] where PUMPNUMBER is an integer constant set to 4.

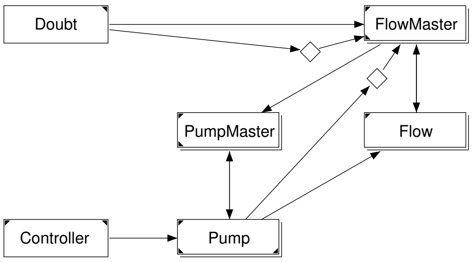
The PumpControl cluster has been designed as reusable software component which accepts alternating opening and closing commands for the pumps. The pump control system supposes that the time between two subsequent pump commands is greater than the maximal delay allowed for the flow sensors to react which is clearly satisfied by the SteamBoilerController system. (If the time between subsequent pump commands is to small, the reaction of the flow sensors becomes unpredictable resulting in an unreliable failure detection.)

When a failure of a pump or a pump control device is detected, the control of the corresponding pump is disabled.

The BoilerControl cluster is informed when the number of enabled pumps changes.

PumpControl has to be informed by the BoilerControl cluster when the correct working of the pumps is doubted.

INTERACTION NET



Two kinds of multiple pulse transmissions occur in cluster PumpControl:
Singular sender and multiple receiver interact always by a one to many broadcast. Multiple senders and receivers with identical index type interact by a one to one transmission. (Senders and receivers with different index type would interact by a restricted many to many broadcast.)

Controller is driven by messages from the BoilerControl cluster and decides about opening and closing the pumps.

Pump signals successful opening or closing of a pump to the supervising instance of Flow. The process detects acknowledge errors which are transmitted to the corresponding PumpMaster.

Flow detects flow errors and signals them to the corresponding FlowMaster.

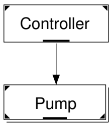
FlowMaster and PumpMaster are failure managers controlling the behaviour of the corresponding Flow and Pump instances respectively.

FlowMaster interprets a flow error either as flow sensor failure or as a pump error depending on the state of Doubt and on the activity of the corresponding pump (inspection). A pump error is transmitted to the corresponding PumpMaster instance. Failures and reparations concerning a particular flow sensor are reported to the corresponding PumpMaster instance responsible for disabling and enabling the pumps.

Doubt is informed by the BoilerControl cluster when the correct working of the pumps is doubted. Arriving doubt messages are signalled to FlowMaster.

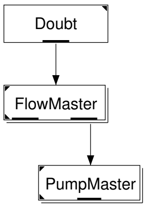
PULSE TRANSLATIONS
SENDER *Controller*
    close_P -> *Pump.close_P*
    open_P -> *Pump.open_P*
SENDER *Doubt*
    inDoubt -> *FlowMaster.inDoubt*
SENDER *Flow*
    flowError -> *FlowMaster.flowError*
SENDER *FlowMaster*
    PCtrl_off -> *Flow.PCtrl_off, PumpMaster.PCtrl_off*
    PCtrl_on -> *Flow.PCtrl_on, PumpMaster.PCtrl_on*
    pumpError -> *PumpMaster.pumpError*
SENDER *Pump*
    pumpError -> *PumpMaster.pumpError*
    pumpStart -> *Flow.pumpStart*
    pumpStop -> *Flow.pumpStop*
SENDER *PumpMaster*
    P_disable -> *Pump.P_disable*
    P_enable -> *Pump.P_enable*

CASCADE *Controller*



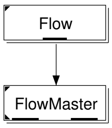
Controller sends opening and closing pulses to all pump instances.

CASCADE *Doubt*



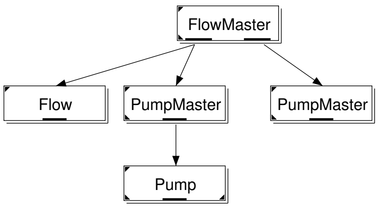
Every received doubt message is signalled to FlowMaster which decides about flow sensor failures and pump errors. A pump error is signalled to PumpMaster.

CASCADE *Flow*



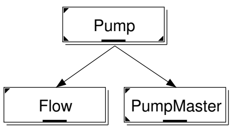
Detected flow errors are signalled to the corresponding FlowMaster.

CASCADE *FlowMaster*



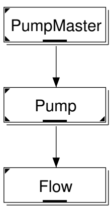
A flow error which is interpreted as flow sensor failure or an occurring reparation event of a flow sensor causes a mode transition of the corresponding Flow instance (left gate). This change of behaviour is also signalled to PumpMaster which must possibly disable or enable the corresponding pump. If a flow error is a pump error it is signalled only to PumpMaster (right gate).

CASCADE *Pump*



Received opening or closing acknowledges of a pump is signalled to the corresponding Flow instance. Acknowledge errors are reported to the corresponding PumpMaster instance.

CASCADE *PumpMaster*



When a pump error has been signalled to PumpMaster the corresponding pump instance is disabled in a chained state transition of the process. The disabled Pump instance must signal the pump stop to the still supervising instance of Flow. Furthermore, when a pump reparation event occurs the corresponding Pump instance must possibly be enabled.

PROCESS Controller

Controller receives opening and closing messages from the BoilerControl cluster. The time between the execution of subsequent pump commands is always greater then the constant delay PUMP_ACK_DELAY. Within this delay the pumps must acknowledge a successful opening or closing.

INQUIRY pumping

```
int pumping (void)
{ return (STATUS.STATE == openedPumps); }
```

EXTENSIONS TIMER

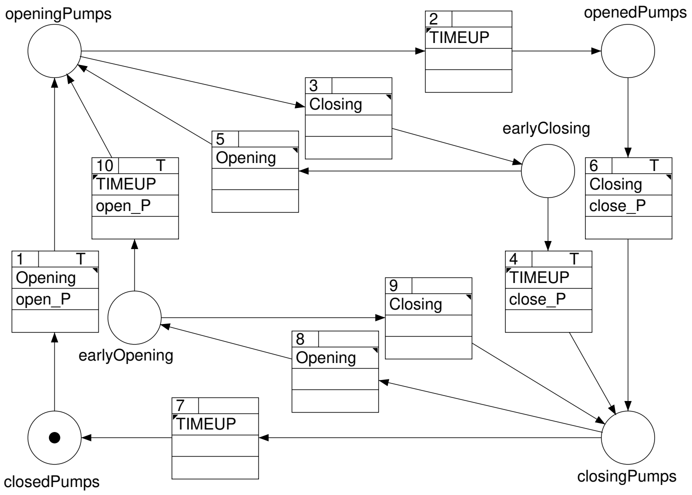
INPORT cmd_in: *pumpMessages*

GATE1 close_P, open_P

DELAY Pump_Acknowledge

```
PUMP_ACK_DELAY * TICKS_PER_SECOND
```

MODE unique



TRANSITIONS 1, 4, 6, 10
SET TIMER *Pump_Acknowledge*

PROCESS Doubt

The process is informed by the BoilerControl cluster whether the correct working of the pumps is doubted. The information is needed by FlowMaster in order to be able to interpret flow errors.

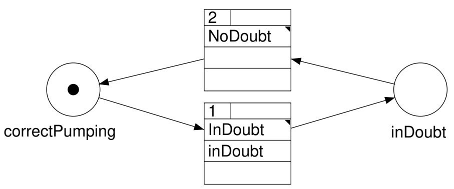
INQUIRY inDoubt

```
int inDoubt (void)
{ return (STATUS.STATE == inDoubt); }
```

INPORT dbt_in: *doubtMessages*

GATE1 inDoubt

MODE unique



PROCESS Flow

The behaviour of the working and broken pump flow sensor are specified by two corresponding modes. The transition structure of the model process has been extended to allow for a timed supervision of the flow of the corresponding pump and for the detection of spontaneously occurring flow events. Flow errors are signalled to the corresponding PumpMaster. When a failure of the pump flow device is recognized by FlowMaster the process mode of Flow is changed to mode broken with no supervision at all.

MULTIPLICITY *PumpRow*

EXTENSIONS TIMER

EVENTS P_FLOW, P_IDLE

INPULSES pumpStart, pumpStop

COMMANDS PCtrl_off, PCtrl_on

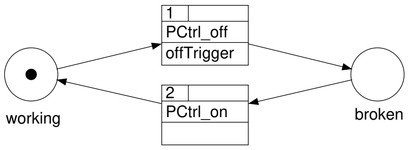
TRIGGERS offTrigger

GATE1 flowError

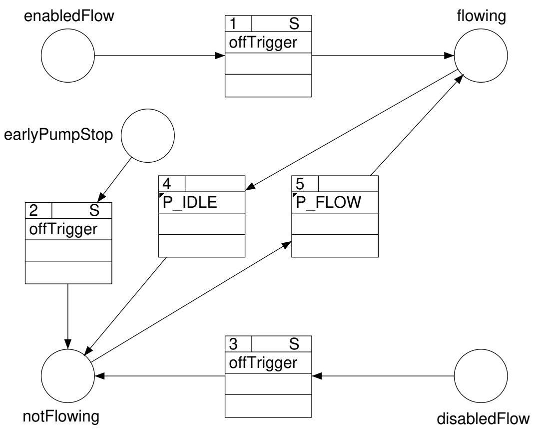
DELAY Decrease_Delay
`FLOW_DECREASE_DELAY * TICKS_PER_SECOND`

DELAY Increase_Delay
`FLOW_INCREASE_DELAY * TICKS_PER_SECOND`

MODERATOR

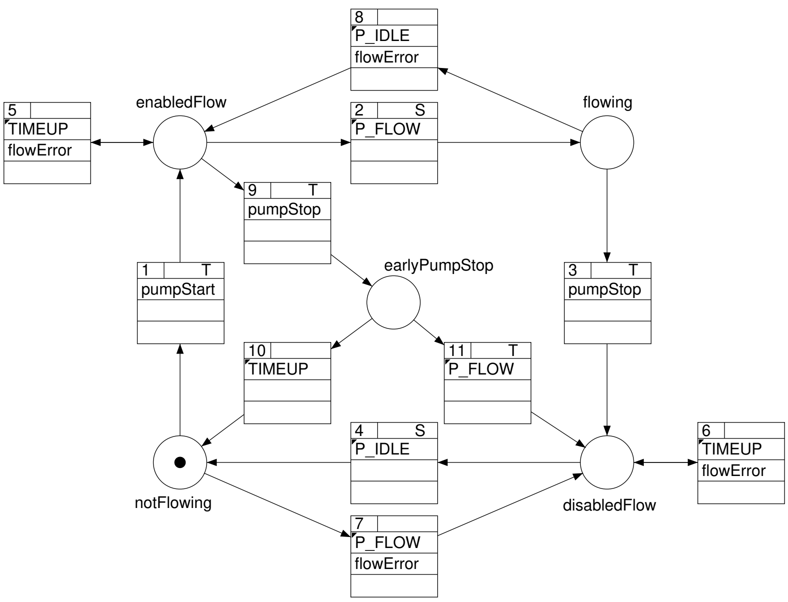


MODE broken



TRANSITIONS 1, 2, 3
STOP TIMER

MODE working



TRANSITION 1
SET TIMER Increase_Delay

TRANSITIONS 2, 4
STOP TIMER

TRANSITIONS 3, 9, 11
SET TIMER Decrease_Delay

The state earlyPumpStop had to be introduced because PumpMaster can recognize at any time a pump failure leading immediately to a pump stop.

PROCESS FlowMaster

FlowMaster controls the behaviour of Flow. The process interprets flow errors detected by Flow as pump control device failures or as pump errors, depending on the correct behaviour of the water level (known by Doubt). If the corresponding pump is already disabled a flow error is necessarily the result of a control device failure.

MULTIPLICITY *PumpRow*

EXTENSIONS TIMER, CHAIN

EVENTS PCtrl_REP

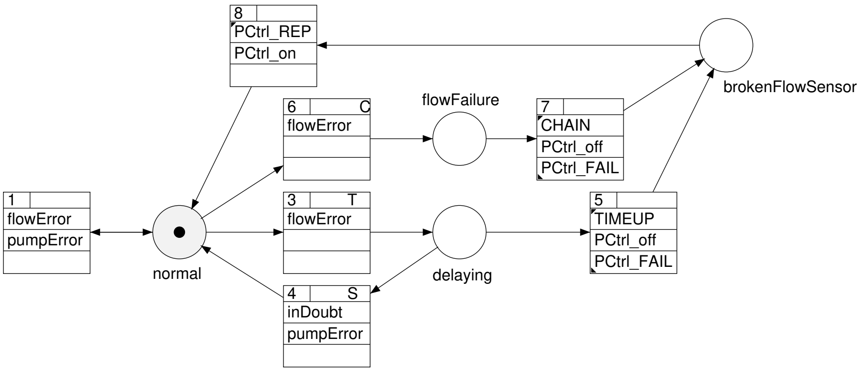
INPULSES flowError, inDoubt

GATE1 PCtrl_off, PCtrl_on
GATE2 pumpError

ACTIONS PCtrl_FAIL

DELAY ErrorPropagation
`PUMPERROR_PROPAGATION_DELAY * TICKS_PER_SECOND`

MODE unique



If an occurring flow error is taken as pump error it is signalled to PumpMaster, if it is a flow failure the disabling of Flow must be chained because Flow has already been activated in the current cluster transition. If the reason for an occurring flow error is not clear, its interpretation is delayed in order to allow the boiler cluster for the detection of incompatible water level dynamics.

BRANCHING STATE *normal* INPULSE *flowError*
    POSTSTATE *flowFailure* IF
    `Pump.disabled(ID.PumpRow)`
    POSTSTATE *normal* IF
    `Doubt.inDoubt() && !Pump.disabled(ID.PumpRow)`
    ELSE POSTSTATE *delaying*

TRANSITION 6
    SET CHAIN

TRANSITION 3
    SET TIMER ErrorPropagation

TRANSITION 4
    STOP TIMER

PROCESS Pump

The behaviour of the enabled or disabled pump is specified by two corresponding modes. The transition structure of the model process has been extended to allow for a timed supervision of the pump acknowledges and for the recognition of invalid occurrences of acknowledge events. When a pump is disabled the process changes to mode disabled, closing if necessary the corresponding pump. Enabling and disabling is always reported by to the BoilerControl cluster by messages written trough outport res_out.

INQUIRY disabled
```
int disabled (index)
{ return (STATUS[index].MODE == enabled); }
```

MULTIPLICITY *PumpRow*

EXTENSIONS TIMER

EVENTS CloseACK, OpenACK

INPULSES close_P, open_P

COMMANDS P_disable, P_enable

TRIGGERS disableTrig

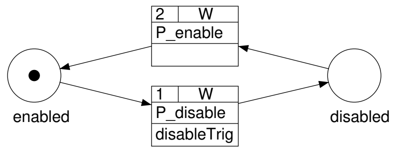
GATE1 pumpError, pumpStart, pumpStop

ACTIONS CLOSE_P, OPEN_P

OUTPORT res_out: *resourceMessages*

DELAY ackDelay
`PUMP_ACK_DELAY * TICKS_PER_SECOND`

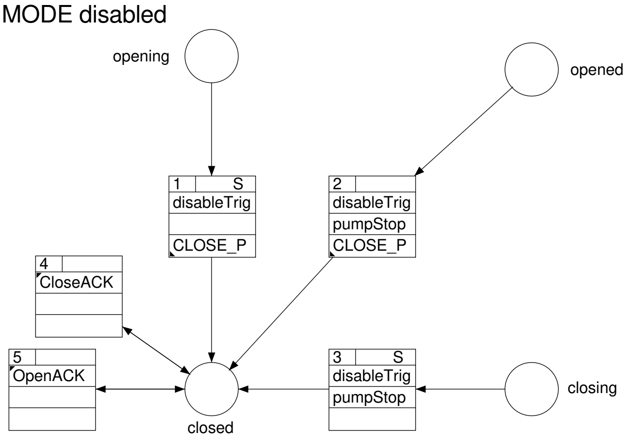
MODERATOR



TRANSITION 1
    OUTPORT *res_out* MESSAGE *Disabled*

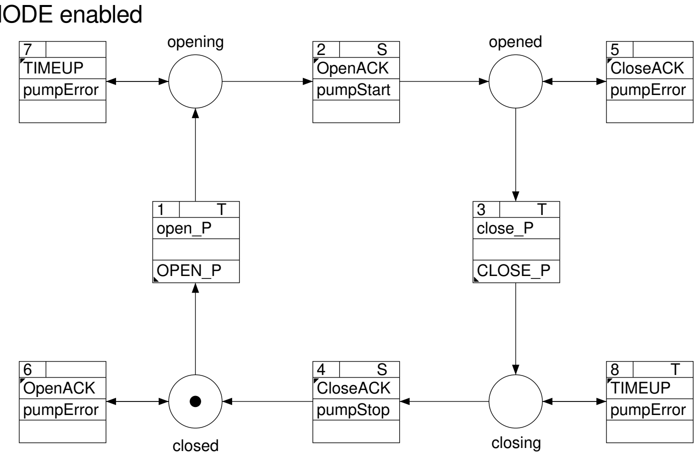
TRANSITION 2
    OUTPORT *res_out* MESSAGE *Enabled*

MODE disabled



TRANSITION 1, 3
    STOP TIMER

MODE enabled



TRANSITIONS 1, 3, 8
    SET TIMER ackDelay

TRANSITIONS 2, 4
    STOP TIMER

PROCESS PumpMaster
PumpMaster controls the behaviour of Pump. For occurring failures and reparations of pumps and pump control devices the corresponding Pump instances are enabled and disabled respectively.

MULTIPLICITY *PumpRow*

EXTENSIONS CHAIN

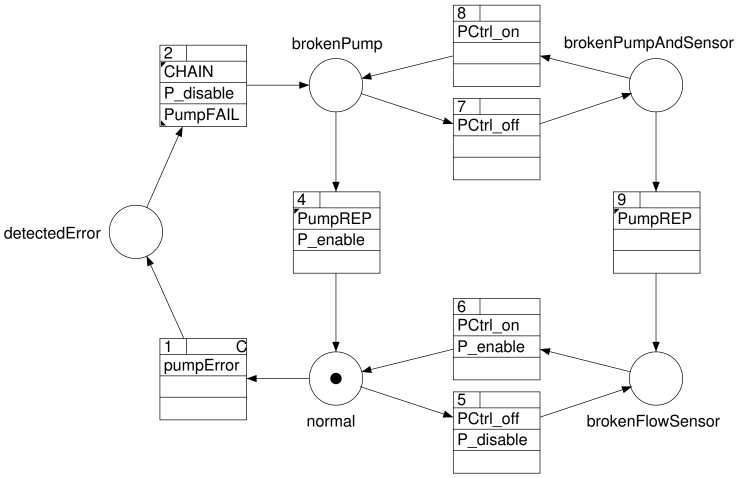
EVENTS PumpREP

INPULSES ackError, PCtrl_off, PCtrl_on, pumpError

GATE1 P_disable, P_enable

ACTIONS PumpFAIL

MODE unique



TRANSITION 1
SET CHAIN

The disabling of Pump for an occurring pump error is chained because Pump has possibly already been activ in the current cluster transition.

6. Experience and Conclusion

The problems encountered during the development of the steam boiler controller specification can be classified as follows:

Control and Supervision

Following the development concepts of the CIP method, the control of the pumps for the proposed modes of operation, the supervision of the critical level limits as well as the verification of the sensor values by calculation could be specified without great difficulty. The problems encountered in a first simulation concerned the error interpretation and the long time behaviour of the controller in Rescue mode.

Error Interpretation

In the simulation of the first version of the controller we realised that an incompatible behaviour of the water level sensor and occurring invalid flow sensor events can depend on each other, which makes the interpretation of errors more difficult than presumed. Furthermore, the unpredictable behaviour of broken pumps does not allow to interpret invalid flow sensor events in the presence of broken pumps.
The problems have been solved in an elaborated specification, introducing postponing of error interpretations (error propagation delay) and supposing fail save pumps. The simulation of the elaborated specification worked correctly, but in Rescue mode the critical water level limits where reached to fast (emergency stop).

Mean time to emergency stop in the presence of broken units

In a third elaboration we introduced a pump model allowing to refine the calculation of the pump flow for opening pumps. Furthermore, we increased the sampling rate leading to a more precise calculation of the water level (second order effects).

Conclusion

A fault tolerant control system must maintain its service in the presence of faults. A main problem when developing such systems is to find an efficient failure management based on a clear error interpretation allowing to maintain the required control as long as possible.

With operational specification models such problems are solved by a kind of evolutionary prototyping. The development process leads to a final specification describing a working solution of the control problem. The CIP method supports this approach strongly due to the constructive use of graphical models resulting in readable and manageable system specifications. Furthermore, the nature of the compositional CIP specification model with its clear interface to the transmission layer allows to generate efficient code which can be linked without modification to the services of the transmission layer of the target system.

References

[1] Abrial J.-R.: Steam-boiler control specification problem. (1994)
http://www.informatik.uni-kiel.de/~procos/dag9523/dag9523.html

[2] Additional information concerning the dynamic behaviour of the steam-boiler. (1995)
http://www.informatik.uni-kiel.de/~procos/dag9523/dag9523.html

[3] CIP Tool®: Graphical specification tool with code C-generators and report writer.
CIP System AG, Postfach 348, CH-4502 Solothurn, Switzerland

[4] MATLAB/Simulink®: Graphical tool for modelling and simulating dynamic systems.
The MathWorks Inc, 24 Prime Park Way, Natik Massachusetts 01760

[5] Fierz H., Müller. H., Netos S.: CIP-Communicating Interacting Processes.
In Gorski J. (ed.) Proceedings SAFECOMP'93, Poznan-Kiekrz, Poland, 1993. Springer Verlag 1993.

[6] Fierz H.: Spezifizieren und Entwickeln mit graphischen Modellen.
Bulletin d. Schweiz. Elektrotechnischen Vereins (1995) 21, pp. 33-39.

[7] Zave P.: The Operational Approach versus the Conventional Approach to Software Development.
Comm. ACM 27 (1984) 2, pp 104-118.

[8] Peterson J.L.: Petri net theory and the modelling of systems.
Prentice-Hall International, Englewood Cliffs N. J, 1981

[9] Harel D.: Statecharts: A Visual Formalism for Complex Systems.
Science of Computer Programming (1987) 8, pp. 231-274.

[10] Cameron J.R. (ed.): JSP and JSD: The Jackson Approach to Software Development.
IEEE Computer Society Press, 1989.

[11] Fierz H.: SCSM - Synchronous Composition of Sequential Machines.
Technical report, Computer Engineering Laboratory, ETH Zürich (1995), submitted for publication.

[12] Fierz H.: Denotational Semantics of CIP.
Technical report, Computer Engineering Laboratory, ETH Zürich (1995), in preparation.