

# *5 Compositional Behavioural Modelling with CIP*

---

The CIP modelling framework supports the process-oriented development of embedded functions. CIP stands for *Communicating Interacting Processes*. CIP models represent composite reactive machines that are constructed graphically with CIP Tool®. Models are built by creating, composing and connecting extended finite state machines. Due to the formal basis of the CIP modelling framework, CIP Tool transforms the constructed reactive machines into software components executable in test environments and on the target systems. In addition, an automated interaction tree viewer supports interaction analysis for models under construction.

CIP models are based on synchronous and asynchronous composition of extended finite state machines. Cooperation of state machines is specified by means of architectural connectors that mediate interaction between the connected state machines. A main characteristic of architectural composition is that components remain untouched when interaction is introduced. The resulting robustness of the modelling components during the development becomes important when models have to be restructured and changed.

This chapter presents the foundation of the CIP modelling framework. We start by developing the notion of state machine composition. Deterministic state machine cooperation can then be achieved by introducing local interaction that is based on the fundamental principle of run-to-completion semantics. In addition, the cluster structure of CIP models is shown to be a generic system partition that arises as soon as synchronous and asynchronous composition of state machines is combined within the same system. The final part of this chapter introduces the interaction types supported by the CIP modelling framework. One simple control problem is solved by five model variants in order to illustrate the nature of the supported interaction mechanisms. The various CIP interaction types are treated in detail in the subsequent Chapters 6 to 9.

## 5.1 An Overview of the Architecture 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.

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.

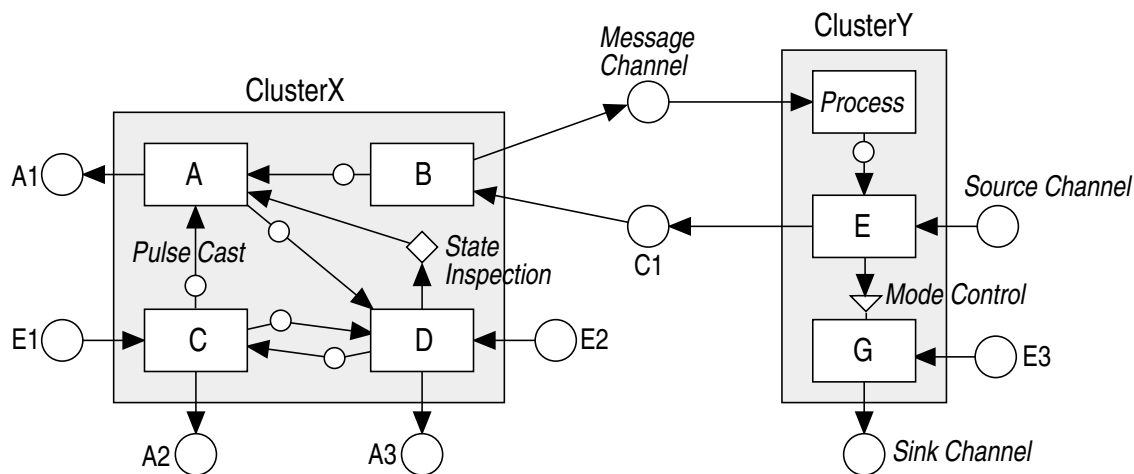


Figure 26: Clusters, processes and interaction connectors of a CIP model

Process cooperation is specified by means of interaction. Figure 26 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 (external 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.

### *5.2 Component Categories: Processes and Clusters*

The CIP modelling framework is based on two generic component categories: processes and clusters. Processes are elementary extended finite state machines while clusters represent synchronous process compositions. Clusters are composed asynchronously to CIP systems.

#### *5.2.1 Synchronous and Asynchronous Composition of State Machines*

In general, state machines can be composed synchronously or asynchronously to larger reactive systems. If we define in addition interaction between such state machines, we obtain synchronously or asynchronously cooperating state machines, respectively. Synchronously cooperating state machines represent parallel components of a sequential reactive machine while asynchronously cooperating state machines model systems of distributed components.

##### *Asynchronous Composition*

Asynchronously composed state machines proceed concurrently at indeterminate relative speed. Synchronisation of asynchronous state machines must be specified explicitly by means of suitable interaction mechanisms. There is no notion of global time and global state.

##### *Synchronous Composition*

Synchronously composed state machines proceed in lock step: that is, all state machines of a composition are involved simultaneously in a common reaction step when an input occurs. In general, an input is a combination of individual inputs for some of the state machines of the composition. The individual inputs are processed simultaneously by the concerned state machines. The other state machines of the composition are idling during the synchronous reaction step: that is, they remain inactive until the reaction step of the composition has terminated. The lock step idea suggests the notion of a global clock, and the state tuple formed of the states of the individual state machines represents the global state of the composition.

An important special case are composition inputs that consist of a single stimulus for a particular state machine only. Such inputs lead to reactions of the composition where all but the triggered state machine remain inactive. Single state machine activation is much simpler to follow than simultaneous reactions of several state machines, especially when interaction between state machines cause sequences of reaction steps. For this reason, the CIP modelling frame-

## 5 Compositional Behavioural Modelling with CIP

---

work provides interaction connectors that cause single state machine activation only (see below).

### *Example*

The following simple example illustrates the difference of synchronous and asynchronous composition of two state machines. The two state machines M1 and M2 consist each of a single state transition only.



Figure 27: The state machines M1 and M2

*Asynchronous Composition of M1 and M2.* The two state machines proceed independently at indeterminate relative speed. An external global observer must therefore expect any input/output sequence that is compatible with the input/output orders produced by the individual state machines, namely “E1 causes A1” and “E2 causes A2”.

For instance, (E1, A1, E2, A2) and (E2, A2, E1, A1) are the possible input/output sequences when one state machine terminates its execution before the other one is started.

Furthermore, it is possible that M2 is triggered when M1 is executing, which leads to the input/output sequences (E1, E2, A2, A1) or (E1, E2, A1, A2). Similarly M1 may be triggered when M2 is executing.

*Remark.* A global observer is supposed to register occurring inputs and outputs much faster than the state machines change their state. In addition, occurring inputs and outputs are registered as sequences. For instance, if E1 and E2 are recognised simultaneously, the observer decides to register either the sequence (E1, E2) or (E2, E1) because the synchronous recognition of E1 and E2 is not relevant for the behaviour of the concurrent state machines.

*Synchronous Composition of M1 and M2.* The two state machines are synchronously composed to a singular state machine that reacts on combination of inputs for M1 and M2. The synchronous composition of M1 and M2 can be viewed as a global behavioural restriction applied to the asynchronous composition of M1 and M2. Regarding the composition as black box, the following input/output sequences can in general occur:

If the two input stimuli  $E1$  and  $E2$  appear as composed input  $E1 * E2$ .  $M1$  and  $M2$  perform their state transition simultaneously, which leads to the input/output sequence  $(E1 * E2, A1 * A2)$ .

(Notation: The products  $E1 * E2$  and  $A1 * A2$  denote synchronous occurrence  $E1$  and  $E2$ , and  $A1$  and  $A2$  respectively.)

If the input  $E1$  occurs before  $E2$  the resulting input/output sequences is  $(E1, A1, E2, A2)$ .  $E2$  before  $E1$  correspondingly leads to  $(E2, A2, E1, A1)$ . For each occurring input, the concerned state machine executes the triggered state transition completely while the other state machine remains inactive.

As already mentioned above, the interaction connectors provided by the CIP modelling framework only lead to single state machine activation of synchronous compositions. If we compose  $M1$  and  $M2$  synchronously within a CIP model, the simultaneous activation of  $M1$  and  $M2$  can therefore not occur. Thus  $(E1, A1, E2, A2)$  and  $(E2, A2, E1, A1)$  are the only possible scenarios.

### *5.2.2 Cooperation by Means of Local Interaction*

Synchronous and asynchronous composition of state machines provide two cinematic schemes that allow to specify interaction between state machines. We use the term *local interaction* to denote the interaction between state machines of a composition while the term *global interaction* is used to denote the interaction with the environment of a composition.

#### *Interaction by Means of Stimuli Transmission*

An obvious way to define interaction between state machines is to use generated output stimuli of one state machine as input stimuli for an other state machine. For a synchronous composition, such stimuli transmission takes place between subsequent reaction steps of the composition; for an asynchronous composition, different stimuli transmission proceed concurrently. In addition, for synchronously composed state machines it is possible to introduce interaction based on state dependencies; that is, the reaction of a state machine may depend on the current states of other state machines of the composition.

In the presence of local interaction, an external input initiates in general a branched chain of stimuli transmissions between the state machines of the composition. Our purpose of inducing chain reactions by means of interaction is to cause specific system reactions in response to external stimuli occurrences. However, an immediately arising difficulty with functional interaction chains is that during the local interaction propagation other external input stimuli can

## 5 Compositional Behavioural Modelling with CIP

occur. Such stimuli can initiate further interaction chains that conflict with the uncompleted one. The result may be an unexpectedly complex system behaviour suffering from race conditions and implicit non-determinism. The resulting system must therefore be expected to be non-deterministic even when the elementary state machines are deterministic.

Let us, for example, extend the synchronous composition of M1 and M2 described above. We suppose that each of the two output stimuli A1 and A2 activate a corresponding device, and that these devices are not allowed to be active simultaneously. We therefore try to prevent the activation of both devices by specifying a suitable interaction between M1 and M2. Both machines are extended by a further output stimulus that informs the other state machine when a device has been activated.

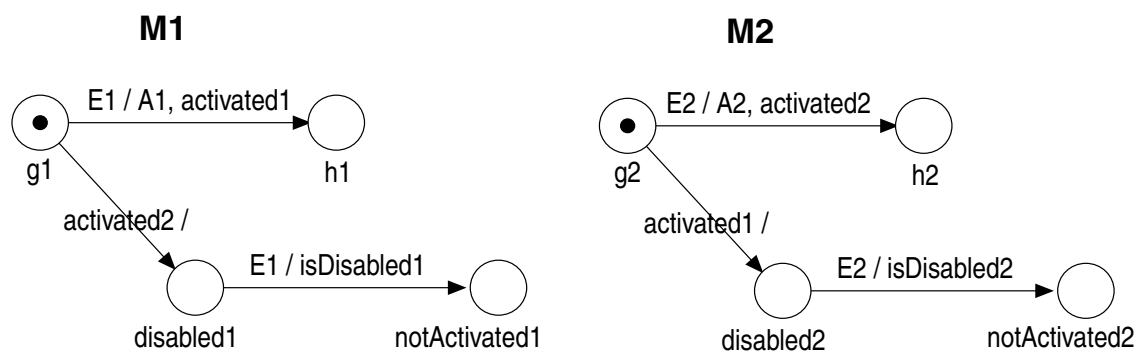


Figure 28: The interacting state machines M1 and M2

Our solution works as follows. If initially M1 is triggered by E1, M1 activates its device by message A1, and it sends the stimulus *activated1* to M2. The transmitted *activated1* stimulus triggers in a second reaction step of the composition M2 to change to the *disabled2* state. A subsequent occurrence of E2 leads to a third reaction step in which M2, instead emitting A2, acknowledges the operator with the *isDisabled2* message.

However, our solution works correct only if the *activated1* stimulus triggers M2 before the external E2 stimulus occurs. If E2 occurs during the first activation of M1, we must expect that E2 is processed before the *activated1* stimulus has been transmitted; thus A2 is emitted and the second device is activated although the first device has been activated in the initial transition of M1.

There are several possibilities to repair our deficient solution. We can extend, for instance, each state machines by suitable transitions that depend on the current state of the other state machine, or we can define additional transitions that are triggered by the synchronous stimuli pairs  $E2^*activated1$  and  $E1^*activated2$ . However, explicit elaboration of interleaved interaction chains

often lead to complex interactions that are difficult to understand. In the following we present a more intelligible approach which solves the encountered difficulty on the semantic level of the model.

### *Run-to-Completion Semantics of Synchronous Compositions*

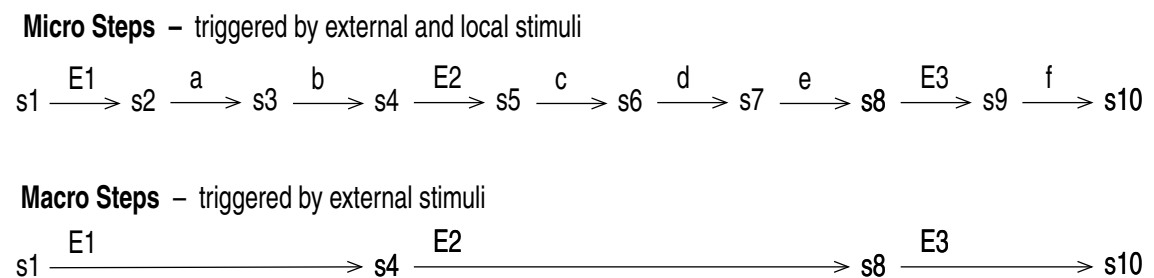
For synchronous compositions, the difficulty of interleaved interaction chains can elegantly be circumvented by introducing *run-to-completion* semantics for local interaction:

#### ***Run-to-completion semantics***

As long as a synchronous state machine composition is to be triggered by locally generated stimuli, external stimulation is disabled.

This behavioural restriction can be applied to synchronous compositions because after every reaction step all components are inactive. Note that with run-to-completion semantics our example above works correctly. As the notion of run-to-completion semantics is crucially based on the notions of global state and clock, it cannot be applied to asynchronous compositions. We must, for instance, indeed expect that when a state machine emits a stimulus to trigger an asynchronous state machine, this state machine is already executing an external stimulus.

The idea of run-to-completion semantics suggest a more extensional description level of synchronously cooperating state machines. When we observe a running synchronous composition we are merely interested in the reached stable states; that is, the states reached after completing the internal interaction chains. By regarding state transition sequences between stable composition states as macro steps, we can abstract from local interactions (Figure 29).



*Figure 29: Micro and macro steps of a synchronous composition (without output)*

A macro step is given by a stable starting state of the composition, by an occurring external input stimulus, by the external outputs generated by the induced chain reaction, and by the reached composition state when the chain reaction has terminated. By describing a synchronous composition by means of macro

steps we obtain a state machine description that hides the local interaction propagation. This view becomes extremely useful when we validate the external reactive behaviour of a synchronous composition, or when we consider its global cooperation with other components of the system.

Run-to-completion semantics reduces drastically the number of possible execution sequences of a synchronous composition. In addition, the restricted behaviour of a composition allows defining local stimuli interaction that leads to deterministic state machine cooperation. Furthermore, even when the individual state machines are not deterministic, we can introduce interaction in the form of state dependencies in such a way that the resulting synchronous composition becomes deterministic.

The notion of run-to-completion semantics assumes that local interaction chains are finite. However, every useful interaction model easily allows constructing cyclic interaction sequences that never terminate. Thus for models based on run-to-completion semantics there remains always the proof obligation of showing that all potential interaction sequences are finite. For time critical applications, it is even not sufficient to guaranty finite interaction chains. In order to satisfy hard real-time conditions the maximal length of local interaction chains must be explicitly bounded. CIP solves this problem by automated verification of the specified interaction: CIP Tool allows constructing synchronous compositions with bounded interaction chains only.

### 5.2.3 The Generic Cluster Structure of a CIP Model

Synchronous state machines composition are well suited to model the reactive behaviour of composite control components due to the notion of run-to-completion semantics. However, as we must support the distributed implementation of reactive components also, we aim to combine synchronous and asynchronous composition of state machines within the same system. CIP models are therefore built by constructing a number of synchronous compositions termed *clusters*, each consisting of a number of state machines termed *processes*. Due to the required run-to-completion semantics of clusters they represent sequential machines that can be composed asynchronously to a distributed reactive machine.

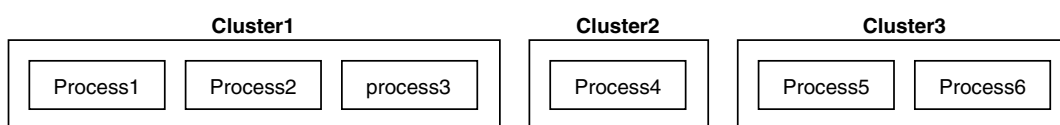


Figure 30: The generic cluster partition of a CIP model

The proposed asynchronous composition of disjoint clusters is in fact a generic compositional system structure that arises as soon as we combine synchronous and asynchronous composition of behavioural entities. The claim becomes clear by the following argument:

If two independent state machines are composed synchronously with a third state machine, the two state machines are automatically synchronised mutually. Thus synchronous composition always leads to disjoint clusters of synchronised state machines. Note that a cluster can consist of a singular state machine. Mathematically, the ‘*synchronised*’ relationship between state machines defines an equivalence relation, and the disjoint clusters represent equivalence classes of that relation.

In addition to the basic cluster structure we can introduce both on the system and on the cluster level further compositional structures. On one hand, we may organise clusters into a hierarchical structure of larger asynchronous subsystems. On the other hand, we can decompose a cluster into subclusters that cooperate synchronously. Both, the asynchronous and the synchronous nesting structure organize systems and clusters in to smaller parts. The purpose of such composition hierarchies is to partition of complex systems into treatable pieces whereas the behavioural semantics of a composition is determined by the underlying generic cluster partition.

### *5.3 CIP Connector Categories: Stimulation and State Sharing*

Our domain oriented description concept for embedded systems claims to describe interaction between modelling components by means of architectural composition. Thus, interaction between processes is to be described by means of specific connectors that mediate interaction between these behavioural components. Processes are correspondingly described as state machines with specific ports that define stimuli or states that are shared with an attached connector. In order to construct an interaction connection between some processes we have to link a connector instance with appropriate ports of these processes. In addition, we must define an interaction relation that specifies how the interaction must relate to the shared interface phenomena.

The CIP modelling framework provides a number of connector types, each supporting a specific interaction mechanism. The basic interaction mechanism

## 5 Compositional Behavioural Modelling with CIP

---

of a connector type addresses the interconnection of plain state machines. For the connection of extended state machines the basic interaction mechanisms are appropriately extended. For example, the transmission of plain stimuli is extended to transport the data of extended stimuli also. Connection configurations are specified graphically. A particular interaction connection is defined by linking a graphical interaction connector symbol with the processes that must participate in the specified interaction.

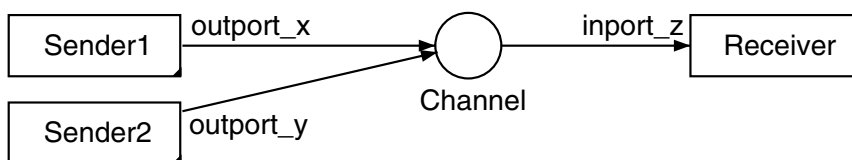
In analogy to the two well known basic communication categories *message passing* and *shared variable access*, we distinguish *stimulation* connectors and *state sharing* connectors. Stimulation connectors generate the control flow of interaction while state sharing connectors allow restricting the behaviour of cooperating state machines by static state dependencies.

### *Stimulation Connectors*

Stimulation connectors transmit stimuli between processes of a CIP model. The processes interconnected with a stimulation connector have either a producer or consumer role. The producer process must be active to initiate an interaction while a consumer process gets activated by the interaction. In addition, the abstract interaction with the physical processes of the system environment (external processes domain) is described by source and sink connectors that model the abstract connection to the external processes.

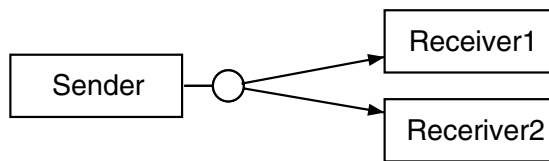
CIP supports two types of stimulation connections: *message passing* is used to transmit stimuli asynchronously between the processes of a CIP system, and between processes and the system environment; *pulse cast* is used within a cluster to cause process chain reactions that run to completion.

#### MESSAGE PASSING INTERACTION



A channel connects one or several sender processes to one receiver process. The connected processes can belong to any cluster of a CIP system. The transmitted stimuli are termed *messages*. Messages sent to a channel are sequentially buffered (fifo). The receiver process is triggered only when its cluster has run to completion. Source and sink channels are channels that transmit messages from and to the system environment.

### PULSE CAST INTERACTION

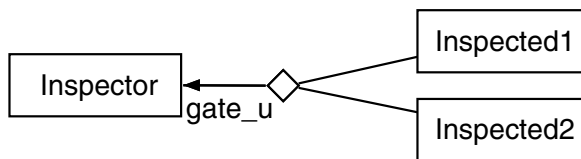


A pulse cast connection links one sender process to one or several receiver processes of the same cluster. The transmitted stimuli are termed *pulses*. A pulse emitted by the sender process triggers those listeners that await the pulse.

### *State Sharing Connectors*

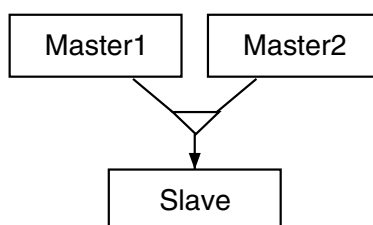
State sharing connectors allow to specify process reactions that depend on the states and variables of other processes of the same cluster. State sharing interaction supposes a notion of global state, thus state sharing has a meaning between processes of the same cluster only. State sharing interaction takes place when the depending process is active.

### STATE INSPECTION INTERACTION



A state inspection connection links one inspecting process (Inspector) to one or more inspected processes of the same cluster. When the inspecting process is stimulated, its reaction can depend on the current states and variables of the inspected processes.

### MODE CONTROL INTERACTION



The behaviour of a process can be specified by several alternative modes. The active mode is determined by a mode control connection. The controlling processes are termed masters, the controlled process is termed slave. The enabled slave mode is determined by the current states of the connected masters.

### 5.4 An Introduction to CIP Models

We illustrate the nature of the various CIP interaction mechanisms by solving a simple control problem – *the Moving Sprinkler System* – in five different ways. The solution samples will show how interaction connections of CIP models are described. CIP models are constructed with CIP Tool, and model descriptions are generated automatically as model reports. All CIP model descriptions in this book are CIP Tool model reports.

In addition, the presented model variants show how the used interaction type affects the design of the state transition structures of the involved processes. In principle, a cluster can always be described by a singular state machine, although for real problems such an approach is not feasible due to the arising state explosion problem. Using a number of interacting state machines allows constructing modular solutions that are understandable, and that can easily be changed and extended.

The five model variants are numbered from A to E. Variant A solves the stated problem by a singular process. The variants B, C, D and E consist each of two interacting processes. Variant B uses *pulse cast* interaction only to solve the problem. Variant C replaces a part of the pulse cast interaction of variant B by *state inspection*. The result is a concise CIP model with simple state transition structures. Variant D replaces the state inspection interaction of variant C by *mode control* interaction. Using different modes to describe the behaviour of the simple control system turns out to be somewhat artificial. Nevertheless, the mode control variant shows nicely how state dependencies can be described on a higher level of abstraction. For completeness, we solve the control problem in the final variant E by replacing the pulse cast interaction of variant B by *asynchronous message passing*. This variant is useful only, if the model is to be implemented on two different processors.

Due to the run-to-completion semantics of CIP clusters, the external model behaviour of the three variants B, C and D are equivalent to the single state machine solution A. That is, executed as black box there is no way to find out which one of the four model variant is running. In variant E the two processes cooperate asynchronously, and run to completion-semantics cannot be enforced in general. The external model behaviour therefore becomes more general. If the model is implemented on a single processor, run-to-completion semantics can be enforced by giving to the internal channel communication higher priority than to the source channels.

The model variants presented in this section are commented briefly only. You will learn more about the four interaction types supported by the CIP model-

ling framework in the following chapters. However, we recommend to compare the model variants again later on, when you have got more experience with CIP models.

### *The Moving Sprinkler System – MSS*

The sprinkler system is used to water uniformly the plants of a rectangular seedling bed. A sprinkler bar is fixed on a car that can be moved on a track by a motor. Two switch sensors are used to detect reaching the back and the front end of the track respectively. The nozzles of the sprinkler bar are connected by a flexible tube to a pump that delivers water in a low or a high pressure mode. A push button serves to command the system, and an acoustic signal (beep) can be used to notify invalid user operations.

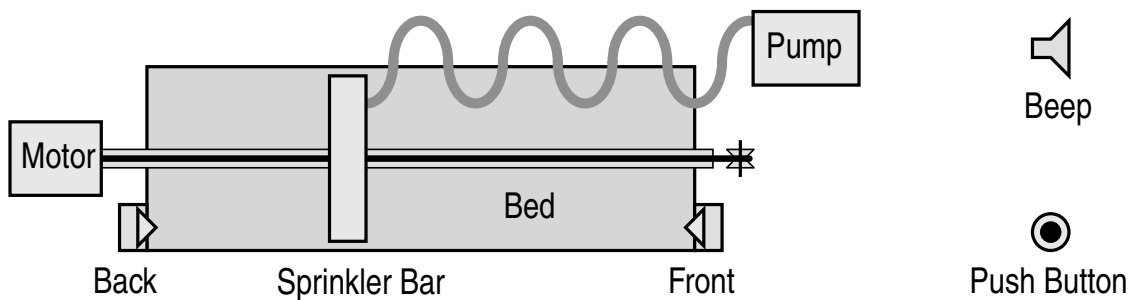


Figure 31: The moving sprinkler (view from above)

#### *Functional Requirement*

When the system is started by pushing the button, the sprinkler begins to move back and forth, and the pump is activated in the low pressure mode. A second button push causes the pump to work in the high pressure mode. Sprinkling is stopped by pressing the button a third time; that is, before the pump is turned off, the sprinkler car must continue its motion until it reaches the back end of the track. If the button is pushed again before the sprinkler car is at rest, the system responds by an acoustic signal (beep) to notify the invalid user action.

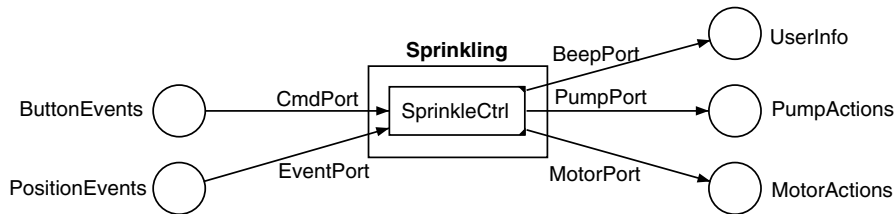
#### *A. The Single PROCESS Solution*

The single process solution describes the functional behaviour of the control system by a single state machine. The state machine is connected to the environment by means of *source* and *sink channels*. Each source channel is linked to an input port, and each sink channel to an output port of the *SprinkleCtrl* process. The stimuli transmitted by channels are called *messages*.

## 5 Compositional Behavioural Modelling with CIP

### SYSTEM MSS\_OneProcess

COMMUNICATION NET SourcesAndSinks



CHANNEL ButtonEvents  
MESSAGES Push

CHANNEL PositionEvents  
MESSAGES AtBack, AtFront

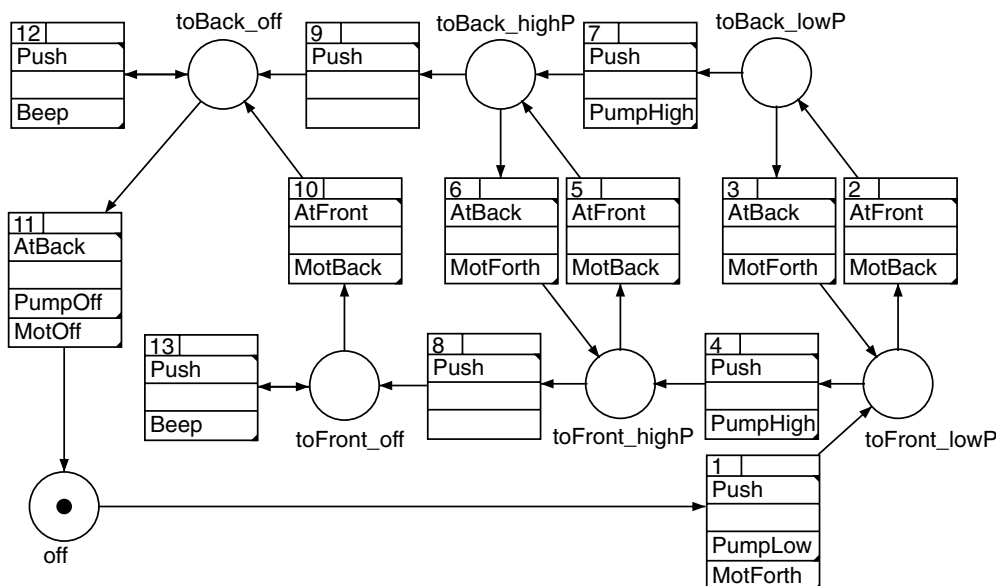
CHANNEL MotorActions  
MESSAGES MotBack, MotForth, Stop

CHANNEL PumpActionst  
MESSAGES PumpHigh, PumpLow, PumpOff

CHANNEL UserInfo  
MESSAGES Beep

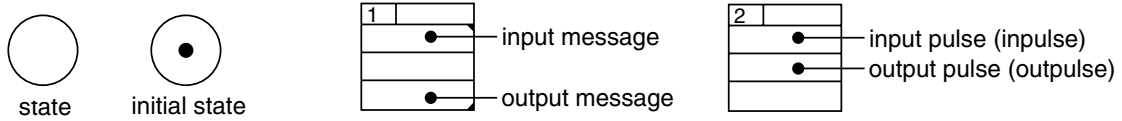
CLUSTER Sprinkling

PROCESS SprinkleCtrl



The messages transported by a channel must correspond one-to-one to the messages specified by the connected process port. For convenience, we use equal names for corresponding channel and port messages. Furthermore, to keep the description of the presented model variants short, we omit the explicit message port descriptions of the specified processes.

*Graphical Notation for States and Transitions*



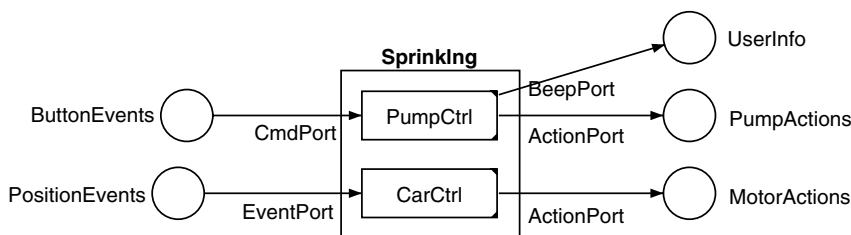
States are represented graphically by circles, transitions by boxes that are each connected to a pre and a post state. The transition number identifies the transition. The symbol in the first field below the transition number designates the input stimulus that triggers the transition. This is either a message of an input port (channel interaction), or an input pulse cast by an other process (pulse cast interaction). The second field may designate an output pulse that is to be emitted when the transition fires. The third and possibly further fields may designate messages of different output ports. Note that the small corner mark of a symbol field indicates that the contained symbol designates a message of an input port (upper corner) or an output port (lower corner).

The state transition structure of our *SprinkleCtrl* process contains message symbols only (no pulse cast interaction). The style of various state names indicate that these states reflect combination of external process states, for example the state *toFront\_lowP* indicates that the car is moving forwards and that the pump works in the low pressure mode. The behaviour of the presented solutions can easily be validated by animating the state machine symbolically. For instance, if the first *Push* message triggers transition 1, the emitted *PumpLow message* causes the pump to deliver water in the low pressure mode, and the emitted *MotFront* message causes the motor to move the sprinkler car in the forward direction.

*Common Specification of Source and Sink Channels of the Variants B to D*

The following three solutions B to D consist each of a single cluster containing two processes termed *PumpCtrl* and *CarCtrl*. In all three variants these two processes are equally linked to the source and sink channels already used in the previous solution. We therefore describe the connection of the *PumpCtrl* and *CarCtrl* processes to these channels in advance and omit the *SourcesAndSinks* net in the following model descriptions.

COMMUNICATIOIN NET SourcesAndSinks

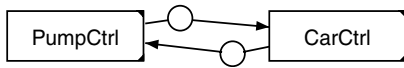


## B. The PULSE CAST Solution

### SYSTEM MSS\_PulseCast

CLUSTER Sprinkling

PULSE CAST NET



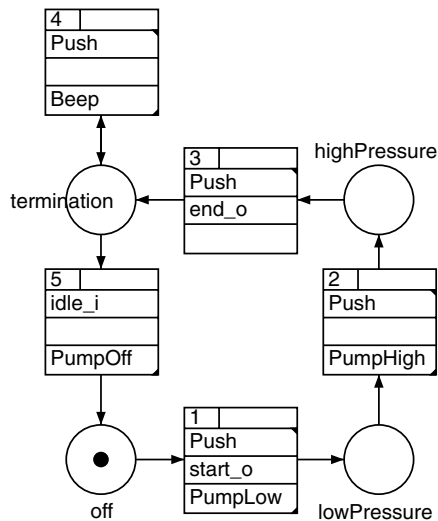
PULSE TRANSLATIONS

SENDER PumpCtrl  
 start\_o -> CarCtrl.start\_i  
 end\_o -> CarCtrl.end\_i

SENDER CarCtrl  
 idle\_o -> PumpCtrl.idle\_i

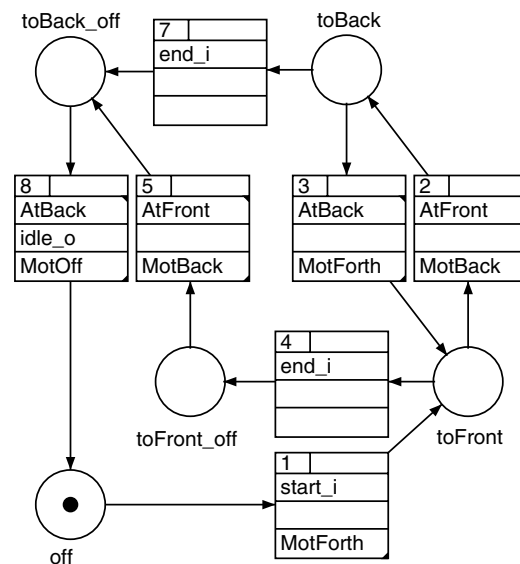
PROCESS PumpCtrl

INPUTS idle\_i  
 OUTPUTS start\_o, end\_o



PROCESS CarCtrl

INPUTS start\_i, end\_i  
 OUTPUTS idle\_o



The system is modelled by two processes of a single cluster. The *PumpCtrl* process reacts on button pushes and controls the pump correspondingly while the *CarCtrl* process reacts on the end position events *AtFront* and *AtBack* to control the car motor. The necessary cooperation of the two processes is caused by pulse cast interaction. Input pulses are called *inpulses* while output pulses are called *outpulses*. Both processes have a unique inpulse port termed *INPUTSES* and a unique outpulse port termed *OUTPUTSES*.

The pulse cast interaction is defined graphically in the PULSE CAST NET. For each of the two connectors there is a PULSE TRANSLATION defined that relates outpulses to inpulses. In general, a pulse cast connection can be linked to more than one receiver process.

In our example we have used for the names of related outpulses and inpulses a common identifier with different postfixes. Outpulses are postfixed with *\_o* and inpulses with *\_i*, such as *idle\_o* and *idle\_i*. We applied this naming rule to explicitly demonstrate that outpulses and inpulses represent different modelling elements. (In practice, many CIP users use equal names for related pulses.)

The first pulse cast takes place when the system is started by the first button push. The reacting *PumpCtrl* process activates the pump in the low pressure mode and casts the *start\_o* outpulse. The listening *CarCtrl* process reacts on the occurring *start\_i* inpulse by activating the motor in the forward direction. The two state transitions of the two processes occur in two subsequent reaction steps of the cluster (synchronous composition). Due to the run-to-completion semantics of clusters, the two process transitions are executed before the next external message is accepted. The resulting macro step of the cluster has externally the same effect as the first state transition of the previous one process solution (A). You can animate the pulse cast solution by marking the current states of the two processes with tokens, and you will see that the observable external behaviour of the pulse cast and the one process solution are equivalent.

### C. The *STATE INSPECTION* Solution

Let us again consider the pulse cast solution. The occurring *end\_i* inpulse of the *CarCtrl* process commands this process to terminate the car motion. Thus the *CarCtrl* process must remember the received *end\_i* inpulse when the next *AtBack* message occurs. The *CarCtrl* process therefore changes to the *toBack\_off* or *toFront\_off* state when the *end\_i* inpulse is received. By using state inspection interaction this history dependent process behaviour can be modelled by a direct dependency on the *PumpCtrl* state. In the state inspection solution presented below, the *end\_o* outpulse, the *end\_i* inpulse, as well as the *toBack\_off* and the *toFront\_off* states have become obsolete and have thus been removed.

In order to model the discussed *PumpCtrl* state dependency directly, we have introduced the gate *working* of the *CarCtrl* process. A process gate represents a boolean interaction point that can be connected by means of a state inspection connector to the state of one or more processes of the same cluster. The *working* gate is therefore connected in the INSPECTION NET to the *PumpCtrl* process. The arrow of the connector points to the dependent process.

The boolean value (TRUE/FALSE) of a gate depends on the current states of the connected processes; that is, the interaction relation is a boolean function which can be specified by the set of process states (or state tuples) that produce the true value. This state set is called TRUTH TABLE. For the connected *working*

## 5 Compositional Behavioural Modelling with CIP

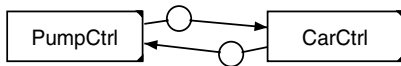
gate, the set states that deliver the TRUE value consists of the *highPressure* and the *lowPressure* state of the *PumpCtrl* process.

The *working* gate is used as a condition to resolve the non-deterministic behaviour in the state *toBack* (marked grey by the tool) of the *CarCtrl* process: because we removed the states *toBack\_off* and *toFront\_off* in this process, there are two state transitions enabled when the *AtBack* message occurs. The non-determinism is resolved by enabling these transitions in dependence on the current *PumpCtrl* state. The GATE ASSIGNMENT below the *CarCtrl* transition structure specifies that transition 3 (to the *toFront* state) fires if *working* is true, and that transition 4 (to the *off* state) is performed otherwise.

### SYSTEM MSS\_StateInspection

CLUSTER Sprinkling

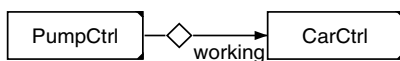
PULSE CAST NET



PULSE TRANSLATIONS

SENDER PumpCtrl  
start\_o -> CarCtrl.start\_i  
SENDER CarCtrl  
idle\_o -> PumpCtrl.idle\_i

INSPECTION NET

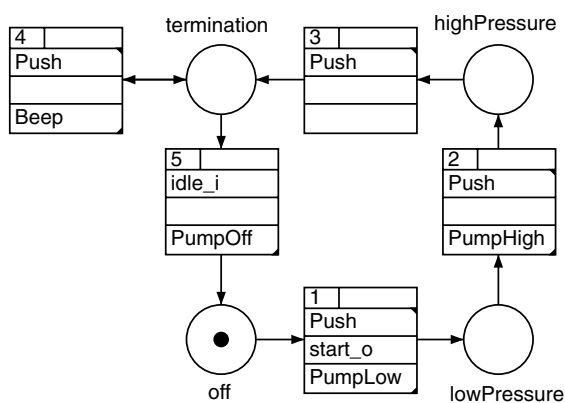


TRUTH TABLES

INSPECTOR CarCtrl GATE working  
RESPONDER PumpCtrl  
TRUE <- highPressure, lowPressure

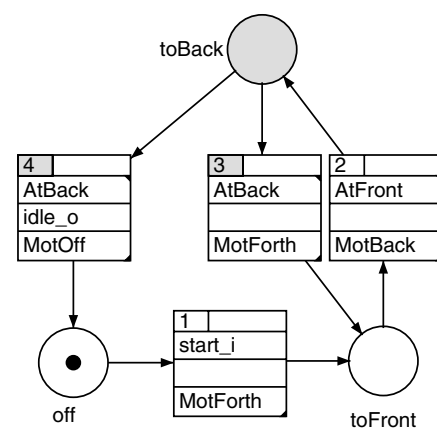
PROCESS PumpCtrl

INPUTS idle\_i  
OUTPUTS start\_o



PROCESS CarCtrl

INPUTS start\_i  
OUTPUTS idle\_o  
GATE working



GATE ASSIGNMENT

STATE toBack MESSAGE AtBack  
TRANSITION 3 / working  
TRANSITION 4 / ELSE\_

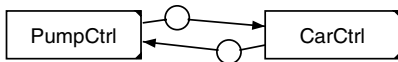
*D. The MODE CONTROL Solution*

The mode control solution describes the behaviour of the *CarCtrl* process by two alternative modes. The state inspection interaction of the previous model has been replaced by mode control interaction.

**SYSTEM MSS\_ModeControl**

CLUSTER Sprinkling

PULSE CAST NET

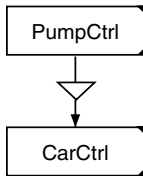


PULSE TRANSLATIONS

SENDER PumpCtrl  
start\_o -> CarCtrl.start\_i

SENDER CarCtrl  
idle\_o -> PumpCtrl.idle\_i

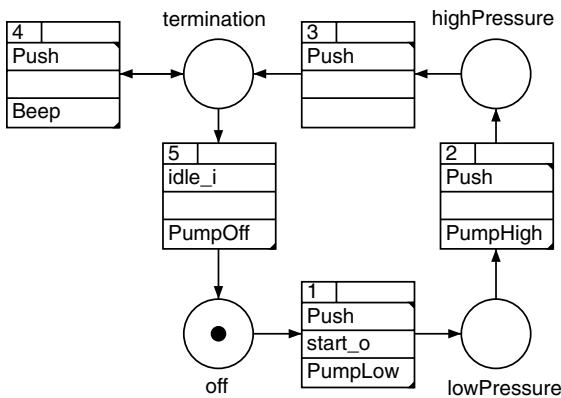
MODE CONTROL NET



MODE SETTING

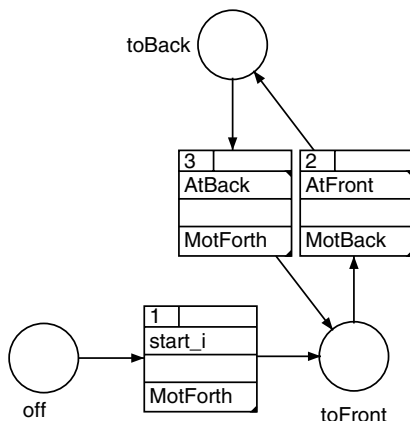
SLAVE CarCtrl  
MASTER PumpCtrl  
working <- highPressure, lowPressure  
terminating <-off, termination

PROCESS PumpCtrl

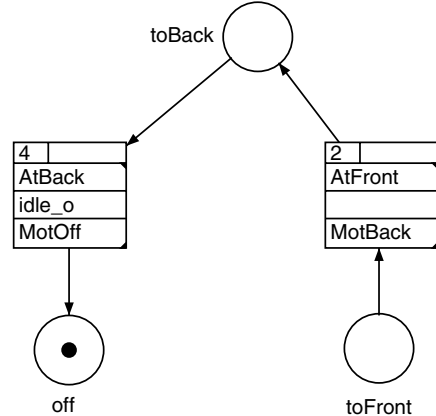


PROCESS PumpCtrl

MODE working



MODE terminating



The behaviour of a CIP process can in general be described by several alternative modes. Each mode is specified by a state transition structure that is based on the states, input stimuli and output stimuli of the process. In our example, the *CarCtrl* process has two modes. The *working* mode describes the process behaviour when the car is continuously moved back and forth while the *terminating* mode specifies the behaviour when the system has to be stopped.

At any time, one mode only of a process is enabled. The enabled mode is determined by the current states of one or more processes of the same cluster. The mode controlled process is termed *slave* while the controlling processes are termed *masters*. The masters for a slave are defined in the MODE CONTROL NET, by connecting master and slave processes by means of a mode control connector.

In the MODE CONTROL NET of our example the *PumpCtrl* process is defined as master for the *CarCtrl* process. The corresponding interaction relation is a function from the master states to the slave modes. In our example this MODE SETTING function defines the *working* mode to be enabled when the *PumpCtrl* process is in the *highPressure* or in the *lowPressure* state. For the other *PumpCtrl* states the *terminating* mode is enabled.

Note that the state inspection and the mode control solution define the same state dependency of the *CarCtrl* process. The difference of the two solutions is how the state dependency is used. In the state inspection solution, the state dependency comes into play only when the *CarCtrl* process must leave the *toBack* state (resolved non-determinism) whereas in the mode control solution we design the process behaviour by two deterministic modes that are alternatively enabled in dependence on the current *PumpCtrl* state.

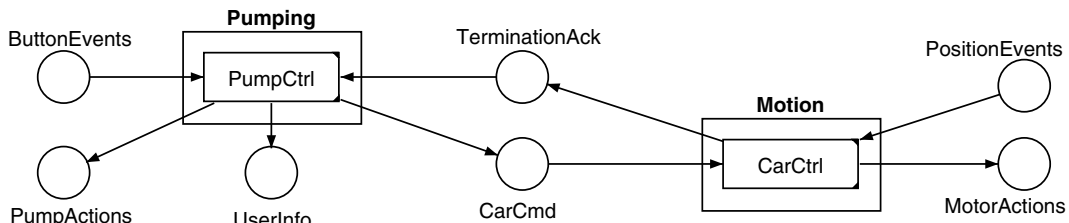
Let us animate the mode control solution. When the system is started by a first button push, both mode control and pulse cast interaction come into play. The occurring *Push* message triggers the *PumpCtrl* process which emits in its transition to the *lowPressure* state the *start\_o* outpulse. The performed state change also changes the enabled mode of the *CarCtrl* process: the terminating mode is disabled and the working mode is enabled. Note that changing the enabled mode does not change the current state of the *CarCtrl* process! In a subsequent cluster micro step, the emitted *start\_o* outpulse is cast as *start\_i* inpulse to the *CarCtrl* process. The occurring inpulse triggers in the *working* mode of the *CarCtrl* process the transition from the *off* to the *toFront* state. Again, due to the run-to competition semantics, the external model behaviour is equivalent to the behaviour of the previous solutions: the occurring *Start* message causes the *PumpLow* and *MotFront* output messages.

*E. The COMMUNICATION Solution*

We give a last variant solution based on asynchronous process cooperation: that is, the *PumpCtrl* and the *CarCtrl* process are located in different clusters. Interaction between clusters is defined by means of channel communication (asynchronous message passing). In the *MSS\_Net* we define in addition to the existing source and sink channels the local *CarCmd* and *TerminationAck* channels. The channel messages correspond to the pulses of the pulse cast solution. For the processes we used process copies of the pulse cast solution. Pulses have been replaced by port messages.

**SYSTEM MSS\_Communication**

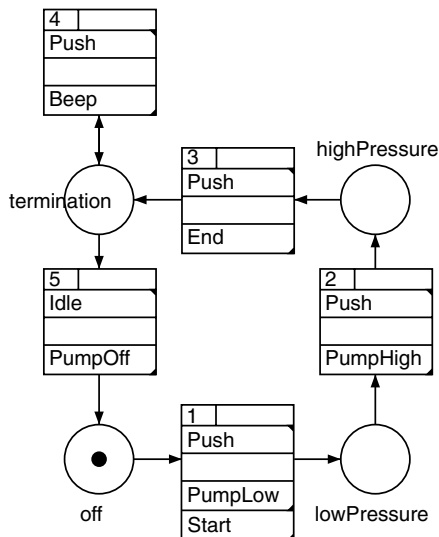
COMMUNICATION NET MSS\_Net



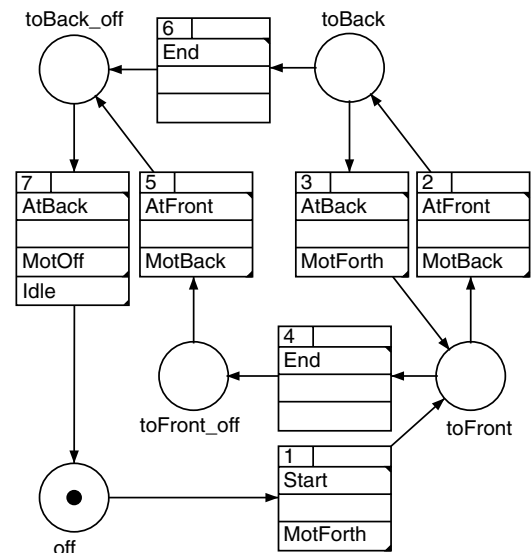
CHANNEL CarCmd  
MESSAGES Start, End

CHANNEL TerminationAck  
MESSAGES Idle

PROCESS PumpCtrl



PROCESS CarCtrl



The model behaviour is more general than the behaviour of the previous solutions because the two processes run concurrently. For instance, a second *Push* message may trigger the *PumpCtrl* process before the *CarCtrl* process receives the *Start* message. As the latency of message passing and cluster reaction is very short with respect to the observed physical times, this solution works correctly too. However, asynchronous cooperation is in general the reason of race conditions and non-determinism that lead to unexpected system behaviour.

