

# *3 Problem Decomposition for Embedded Systems*

---

The main task of an embedded system is sensing and controlling a collection of active external processes. The essential problem to be solved is to construct control functions that bring about a specific required behaviour of the external processes. In addition to the functional problem, there is always the difficult task of accessing the peripheral devices that are needed to connect the external processes with the embedded system.

Domain-oriented development of embedded systems is therefore based on a decomposition of the embedded system problem into a functional and a connection problem. The problem decomposition leads to a layered separation of the functional solution and its connection to the external processes. We elaborate the problem decomposition in this chapter by analysing the generic problem domain structure of embedded systems. The problem analysis is based on the notion of problem frames recently developed by Michael Jackson [REF].

The proposed problem decomposition is fundamental for the CIP method presented in the later chapters. CIP supports the domain-oriented development of the functional solution. Embedded functions are specified by means of executable behavioural models. Furthermore, the development steps stipulated by the CIP method base the construction of CIP models on behavioural correspondences with the external processes. The model-based description technique and the notion of behavioural correspondences are introduced in the chapter about process-orientation (chapter 4).

The connection of the software generated from behavioural models to the peripheral devices is developed with tools and techniques adapted to the applied interface technology. As the nature of the connection problem crucially depends on the characteristics of the used peripheral devices we cannot expect a generic approach to solve this problem. Nevertheless, we shall give in the implementation chapter some patterns and examples that show how the connection problem can be tackled.

### 3.1 Separation of Functionality and Connection

Much of the complexity of embedded systems is associated with controlling special peripheral devices with specific and restrictive interfaces. Therefore, embedded software is usually divided into two groups of components: one group consists of device interface modules containing the device-dependent code; the other group contains the device-independent control code termed reactive machine (see Figure 4). The main purpose of this decomposition is information hiding and ease of peripheral device change. Although the device interface modules generally make the rest of the software simpler, their interfaces are usually the result of an ad hoc design process, and they often fail to encapsulate all device details.

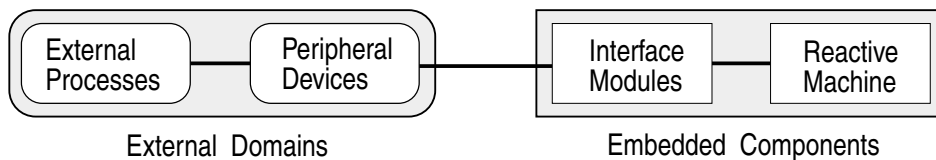


Figure 4: External domains and embedded components

Analysing the structure and properties of the general embedded system problem allows us to look for a generic problem decomposition, rather than for a decomposition of the particular solution. The decomposition concept formulated in this section claims to base the functional solution on an abstract connection to the controlled processes while the connection of the control functions to the peripheral devices is treated as implementation of the abstract connection. The claimed concept is the result of the problem analysis presented in the subsequent sections 3.2 and 3.3. The proposed problem decomposition leads to a generic software architecture that separates the embedded software into a functional and a connection layer. The layered embedded architecture is presented in the final part of this section.

#### 3.1.1 Virtual Devices versus Problem Decomposition

One characteristic of embedded real-time systems is their close relationship with peripheral devices. The interfaces of these devices come in all flavors and may lead to a plethora of different value encodings and timing properties in a single system. Aside from complicating the initial development, the necessary changes in systems that are inadequately structured can be severe if one of these complex and restrictive external interfaces is replaced.

This problem was recognized early on and has led to the concept of abstract interfaces, an attempt to isolate a system's functional code from all interface

### 3.1 Separation of Functionality and Connection

---

device details. An abstract interface may consist of interfaces of virtual devices with device-like capabilities that are partly implemented in software. Virtual devices simplify the rest of the software, liberating it from interface particulars.

A virtual device consists of a device and an interface module that translates between the device interface and the abstract interface of the virtual device. The abstract interface for a given type of device describes the common aspects of devices of that type. Finding a good abstract interface is a difficult task that requires an iterative design process. There remains, on one hand, always a risk that a virtual device based on a deficient abstraction may need to be changed at its interface when the device is replaced. On the other hand, adaptation of the internal virtual device code may become necessary even when the device remains the same; for example, if device-independent functionality that was misplaced into the interface module needs to be changed.

However, our goal is to define an abstract interface that supports the development of the functional solution by means of a behavioural model that reacts on notifications of external processes events. Thus we need an abstract interface that is related to the real external process phenomena that are to be monitored and controlled. For example, if we control an elevator door, the behavioural model must react when the *Opened* event has occurred. Our aim is to completely abstract from the sensor type that is used to detect the *Opened* event.

We shall base the separation of functional and connection concerns on an abstract connection between external processes and functional behavioural models. The abstract connection is determined by the primary purpose of the required embedded system which is monitoring and controlling the external processes. This is in contrast with the virtual device approach which focuses on common characteristics of potential device variants.

#### *3.1.2 Decomposition Concept:*

##### *Layered Separation of Functionality and Connection*

The decomposition concept claims to develop functional embedded solutions on a higher level of abstraction than the connection to the peripheral devices:

Embedded control functions  
and their connection to the system environment  
are to be developed within different layers of interaction.

The problem of constructing an embedded system is decomposed into a functional and a connection problem. Functional components are developed independently of the peripheral devices used to connect external processes and

### 3 Problem Decomposition for Embedded Systems

embedded system. The necessary abstraction is achieved by supposing an abstract connection between external processes and functional components (functional layer). The abstract connection signals occurring process phenomena to the functional components while their reactions cause the abstract connection to act on the external processes.

The implementation of the abstract connection is solved as a problem on its own (connection layer). One part of the real connection between external processes and functional components is given by the installed peripheral devices (hardware connection). This partial connection is to be completed by developing interface modules (software connection) that interconnect peripheral devices and functional components.

#### 3.1.3 Domain-Oriented Architectural Structure

Basically, there are two structures that can determine a system's software architecture, the problem structure and the solution structure. The problem structure plays a fundamental role when subject matter concerns of a system are separated. Such structures relate objects and properties of the real world and are more stable and less ambiguous than solution structures invented during the system development. Structures of isolated subproblems can be used to define a first compositional structure of the solution. The more refined structure of the software-architecture is elaborated along the solution development.

The proposed decomposition concept leads to a layered architecture of the developed embedded system. Figure 5 below shows the connections between the external domains and the embedded components. The conceptual architectural structure results from the decomposition of the general embedded system problem described in this chapter. The *Reactive Machine* depicted in the figure is to be rigorously specified by an executable behavioural model, as proposed by the description concept of domain-orientation.

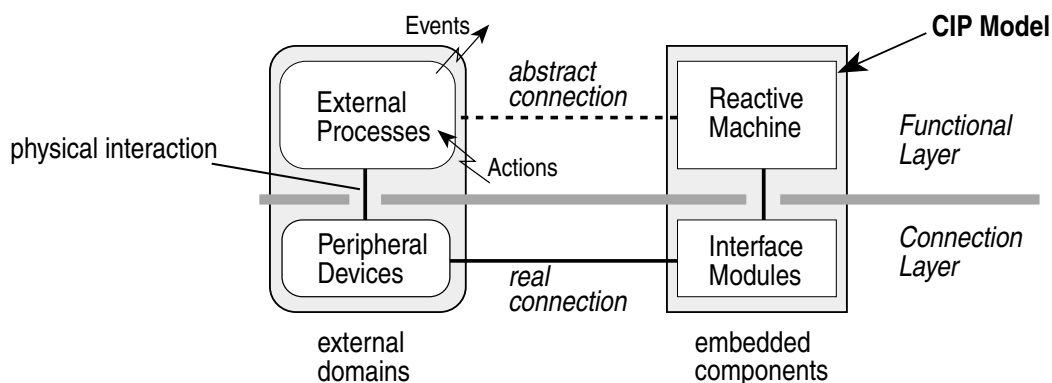


Figure 5: Conceptual embedded system architecture

### 3.1 Separation of Functionality and Connection

---

The construction of the reactive machine model is based on the abstract connection to the external processes. The abstract interface of the external processes consists of collections of events and actions designating instantaneous real world phenomena. Process events are caused by the autonomous dynamics of the external processes. Temporal events occur at specific points in time such as periodic sampling events associated to quasi continuously controlled processes. Actions are process phenomena caused by reactions of the reactive machine. The abstract connection transmits a corresponding message whenever an event has occurred or an action must be produced.

The real connection between external processes and reactive machine consists of peripheral devices (sensors, actuators) and of active interface modules. An interface module extracts events by monitoring sensor interfaces and triggers the reactive machine with corresponding event notifications. In addition, it accepts action commands from the reactive machine and controls the actuators to cause the corresponding process actions.

Domain-oriented embedded system development effects the decomposition of the full problem into a functional and a connection problem right at the inception of the system development. This is not just a question of reducing a large problem to two smaller ones, but of disentangling two problem complexes belonging to different abstraction levels. The functional problem can – due to its isolation from peripheral devices – be solved completely within a suitable modelling framework. But also the connection problem is treated as a software problem on its own. Its solution is developed with specific methods and tools adapted to the applied interface technology. Both problems have their own solutions and corresponding implementations.

A very important benefit of this problem decomposition appears when control functions have to be validated. As the developed software can rarely be tested directly on the target system, it is necessary to use simulation models and specific test beds. The proposed approach allows a functional solution to be partitioned in various ways and to be easily embedded within various test environments.

### *3.2 The Decomposition Problem*

The claimed decomposition concept and the resulting layered embedded architecture is one of many possible approaches to isolate the peripheral device interfaces from the functional solution. Most approaches used in praxis are based on modularisation principles that stipulate how to decompose a particular solution. However, our goal is to find a generic decomposition that is based on the general problem structure. System structures based on problem decomposition are expected to be much more stable during the software life cycle than system structures based on a particular solution.

To find a generic problem decomposition we must analyse the general embedded system problem, i. e. we must identify the problem domains of this problem class and see how these domains are connected.

#### *3.2.1 Problem Frames*

The idea of problem frames has recently been developed by Michael Jackson. The novel approach to problem analysis is presented in his beautiful book *Problem Frames* [REF]. We shall use the problem frame approach to describe the decomposition of the embedded system problem stipulated by our decomposition concept.

A problem frame characterises a class of software development problems. It consists of a collection of given domains providing the subject matter of the computation, of a machine domain representing the machine to be built, and of a requirement imposing conditions on the given domains. The machine domain represents one or more computers specialised by the software. The domains are connected by interfaces, each defined by a particular set of shared phenomena.

#### *Simple Control Problems*

Simple control problems fit to the *Required Behaviour Frame* presented in Jackson's problem frame book. The *Required Behaviour Frame* is one among a set of elementary problem frames. Jackson describes the intuitive notion of the *Required Behaviour Problem* as follows:

“There is some part of the physical world whose behaviour is to be controlled so that it satisfies certain conditions. The problem is to build a machine that will impose that control.”

Here is the problem frame of the *Required Behaviour Problem*:

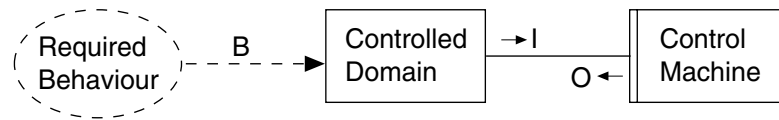


Figure 6: Required Behaviour Frame

The problem frame shown in Figure 6 is a graphical representation of a class of control problems termed *Required Behaviour Problem*. The frame contains the given domain *Controlled Domain* and the machine domain *Control Machine*. The *Controlled Domain* is the part of the real world to be controlled and the *Control Machine* is the machine to be built.

The solid line in the figure represents the interface between the *Controlled Domain* and the *Control Machine*. The interface annotations  $\rightarrow I$  and  $O \leftarrow$  denote two sets of controllable phenomena shared by the *Controlled Domain* and the *Control Machine*. The phenomena of the set  $I$  are controlled by the *Controlled Domain* while the phenomena of the set  $O$  are controlled by the *Control Machine*. These interface phenomena represent states and events shared by the interconnected domains.

The dotted ellipse *Required Behaviour* is the requirement to be brought about by the machine. The conditions and restrictions imposed by the requirement are expressed in terms of the phenomena  $B$  of the *Controlled Domain*.

#### *Control Problem with Operator*

The *Commanded Behaviour Frame* is an extension of the *Required Behaviour Frame*. It has an additional *Operator* domain and an extended emphasis on its requirement.

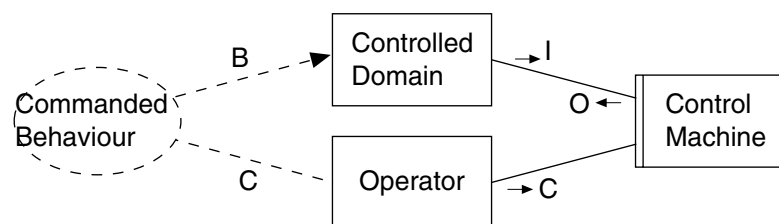


Figure 7: Commanded Behaviour Frame

The *Control Machine* and the *Controlled Domain*, and their phenomena  $B$ ,  $I$  and  $O$  are the same as in the *Required Behaviour Frame*. But now there is an *Operator* issuing commands  $C$ , which appear in the diagram as events shared with the *Control Machine* and controlled by the *Operator*.

### 3 Problem Decomposition for Embedded Systems

The requirement is called *Commanded Behaviour*. It constrains the behaviour of the *Controlled Domain* by describing general rules for its behaviour and specific rules for how it must be controlled in response to the *Operator's* commands  $C$ .

#### 3.2.2 Embedded System Problems

The presented problem frames are too simple to capture embedded system problems. Because of the non-intelligence of the processes to be controlled, establishing connections to the computers represents an intrinsic difficulty in embedded system development. Whether the required process behaviour is attainable at all is depending on the availability of certain physical interactions of sensors and actuators with these processes.

Here is an elaborated problem frame:

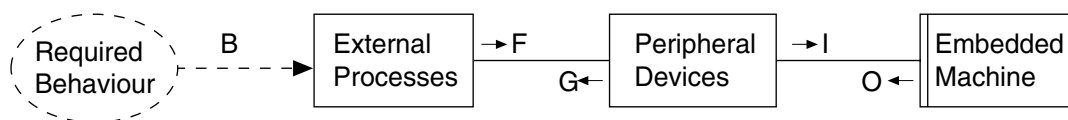
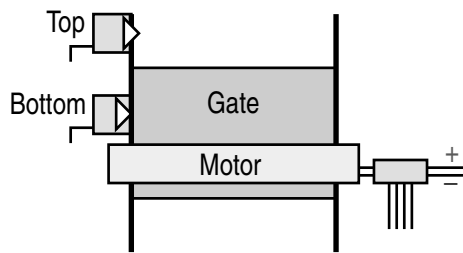


Figure 8: Embedded Control Frame

The *Embedded Control Frame* describes the real world by the two given domains *External Processes* and *Peripheral Devices*. The two domains are interconnected by the two sets  $F$  and  $G$  of shared phenomena expressing physical interaction. The machine to be constructed is now called *Embedded Machine*, and its interface to the real world is described by the set  $I$  and  $O$  of phenomena shared with the *Peripheral Devices*. For simplicity, we shall use this frame also when an operator can affect the embedded system's behaviour. Operators are considered as external processes that are connected to the embedded machine via standard man-machine-interface devices.

In the problem frame we can recognise a basic difficulty. The interface phenomena  $I$  and  $O$  of the *Embedded Machine* are not directly related to the *External Processes* domain, on which the requirements are imposed. The connection between the machine and the chief problem domain is indirect, through *Peripheral Devices* which interpose their properties and behaviour. The gap between the requirement phenomena  $B$  and the specification phenomena  $I$  and  $O$  must be bridged by properties of both given domains, the *External Processes* and the *Peripheral Devices*.

We borrow a simple example of an embedded control problem from Michael Jackson's problem frame book:

*The Sluice Gate Control System*

A small sluice, with a rising and falling gate, is used in a simple irrigation system. A computer system is needed to control the sluice gate: the requirement is that the gate should be held in the fully open position for ten minutes in every three hours and otherwise kept in the fully closed position.

The gate is opened and closed by rotating vertical screws. The screws are driven by a small motor that turns clockwise and anticlockwise, depending on the direction of the applied current. The current can be turned on and off by an electrical relay that is controlled by *on* and *off* pulses. The direction of current is maintained by an electrical polarity switch that is controlled by *neg* and *pos* pulses. There are two mechanical switch sensors that detect the *top* and the *bottom* position of upper edge of the gate: that is, when the top switch is pressed, the gate is fully open, and when the bottom switch is released, the gate is fully closed. The connection to the computer consists of two status lines for the gate sensors and four pulse lines for motor control.

The problem is fitted to the *Embedded Control Frame* by designating the requirement and interface phenomena.

Process states addressed in the requirements:

$$B = \{\text{Open, Shut}\}$$

States shared between external process (sluice gate) and peripheral devices:

$$F = \{\text{Open\_TopSwitchPressed, Shut\_BottomSwitchReleased}\}$$

$$G = \{\text{MotOff\_RelaisClosed, MotRaising\_RelaisOpen\&PolarityNeg, MotLowering\_RelaisOpen\&PolarityPos}\}$$

States and events shared between peripheral devices and embedded machine:

$$I = \{\text{top, bottom}\} \text{ – shared boolean interface variables}$$

$$O = \{\text{on, off, pos, neg}\} \text{ – shared pulses}$$

*Remark on shared phenomena.* A phenomenon shared between domain A and domain B is a phenomenon of both domains. It typically involves a phenomenon of domain A, a phenomenon of domain B and a sharing mechanism that constrains these phenomena to occur together. For instance, the shared state *Open\_TopSwitchPressed* involves the gate state *Open*, the top switch state

### 3 Problem Decomposition for Embedded Systems

---

*TopSwitchPressed* and the mechanical interaction mechanism that synchronizes the mechanical states of the to objects.

When we build the machine, at least three different problems have to be solved. The *Embedded Machine* must interpret the sensor interface variables  $I$  and deduce the sluice gate behaviour, it must define reactions to bring about the required control functionality, and it must initiate the actuator pulses  $O$  to produce according effects on the sluice gate motor. Thus the problem of building an *Embedded Machine* is a composite problem that addresses both functional and connection concerns.

#### 3.2.3 Problem Decomposition

The analysis of a problem class by means of problem frames do not just aid in understanding the relationship of a problem's principle parts. By designing the machine to be built as collection of submachines, and by decomposing problem domains into smaller domains, we are able to isolate specific aspects of the entire task of building a machine. This isolation permits the separate partial development within a limited and clearly defined scope.

Our discussion makes a decomposition of the *Embedded Control Problem* into a functional and a connection problem obvious. The functional problem is to construct a reactive machine that is independent of the installed peripheral devices. The connection problem is to develop an embedded connector which relates interface phenomena of peripheral devices to interface phenomena of the reactive machine.

To get a clue of how the *Embedded Control Problem* is to be decomposed we sketch the expected decomposition of the *Embedded Machine*.

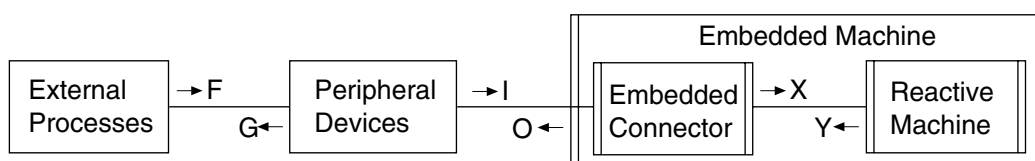


Figure 9: Decomposition of the Embedded Machine

The depicted machine decomposition stipulates to build the *Embedded Machine* by constructing two interconnected submachines: a functional machine termed *Reactive Machine* and a connection machine termed *Embedded Connector*. The *Embedded Connector* interprets the shared sensor phenomena  $I$  and activates the *Reactive Machine* correspondingly. The *Embedded Machine*

must bring about the required system functionality by causing the *Embedded Connector* to control the phenomena  $O$  shared with actuators accordingly.

The crucial question of the desired decomposition is “What is the interface among the two submachines?” or, formulated differently, “What are the phenomena  $X$  and  $Y$ , shared by the *Embedded Connector* and the *Reactive Machine*?”

A domain-oriented choice consists of phenomena that are related in a simple way to phenomena of the *External Processes* domain. Such a choice will allow us to adopt a process-oriented view when we construct the *Reactive Machine*. Our desired result is a machine that depends on the behavioural properties of the external processes only.

### *Shared Process Phenomena Mirrored at the Reactive Machine Interface*

A first obvious idea of a choice rule – not the good one as we will see – consists of mirroring the *External Processes* phenomena shared with the *Peripheral Devices* at the *Reactive Machine* interface. To be exact, we like to simulate, the *External Processes* phenomena that cause the shared physical interaction phenomena  $F$ , and we want the *Reactive Machine* to produce phenomena that correspond to the *External Processes* phenomena created by the shared physical interaction phenomena  $G$ .

Applying the idea to the sluice control example gives for the *Reactive Machine* interface

$X = \{\text{Open, Shut}\}$  as mirrored gate states

$Y = \{\text{MotRaising, MotLowering, MotOff}\}$  as mirrored motor states

We used here for the phenomena of the *Reactive Machine* interface the same names as for the corresponding phenomena of the *External Processes* domain. For instance, the interface state *Open* of the set  $X$  mirrors the real *Open* state of the gate, involved in the shared mechanical state *Open\_TopSwitchPressed*. Or *MotRaising* of the set  $Y$  mirrors the real *MotRaising* state of the gate motor, involved in the shared electrical state *MotRaising\_RelaisOpen&PolarityNeg*.

However, even in our simple example we can see two main shortcomings of the recipe of relating the phenomena of  $X$  and  $Y$  to process phenomena shared with the peripheral devices. First, although *Open* and *Shut* are state phenomena of the *External Processes* domain that are perfectly independent of the *Peripheral Devices* domain, the choice itself depends on the type of installed sensors and actuators. If we replace the gate switches, for instance, by electro-mechanical pulse generators, the rule would lead to an other kind of mirrored process phe-

### 3 Problem Decomposition for Embedded Systems

---

nomena, namely the events *ReachTop* and *ReachBottom*. A further interesting variant is a the sluice gate system where the gate position is monitored by counting sensed rotations of the screws. In general, we must expect complex sensor technologies that base the interaction with the external processes on phenomena that are not directly relevant for the particular control problem. We may for instance need more than one sensor to detect a singular external process event.

Second, the *Reactive Machine* input interface in our example consists of the shared states *Open* and *Shut*. Hence, the *Reactive Machine* must continuously read these states to detect reaching the top or bottom gate position respectively. We consider the task of detecting changes of interface variables as a part of the connection problem (polling, busy wait loops). Although sensors are often based on a type of physical interaction that synchronizes external process states and sensor states, we should not be forced to build reactive machines with shared variable interfaces.

These arguments show that the rule of mirroring the physical peripheral device interaction at the reactive machine interface does not lead to a true domain-oriented decomposition of the embedded system problem. The task of the reactive machine is to bring about requirements that are formulated in terms of certain essential external process phenomena. The choice of the interface of the reactive machine must therefore be guided by the requirements, not by the peripheral devices. To be able to build an intelligible reactive machine, we ought to base its interface on subject matter phenomena that are directly related to the phenomena addressed in the requirements.

### 3.3 The Process-Oriented Problem Decomposition

In order to be able to formulate our problem decomposition independently of the used peripheral devices, we consider the more general problem of constructing a computer-based control system and treat the embedded system problem as a subproblem.

#### 3.3.1 The General Process Control Problem

The solution task of the general control problem consists of interconnecting the controlled processes and the used computers by means of peripheral devices and of developing the required computer software. This broader view will allow us to formulate the functional part of the embedded system problem independently of the installed peripheral devices.

The notion of problem frames was invented to describe the structure of software engineering problems. We adapt the notation for computer-based system engineering problems ad hoc.

Here is the top level frame for the *Process Control Problem*:

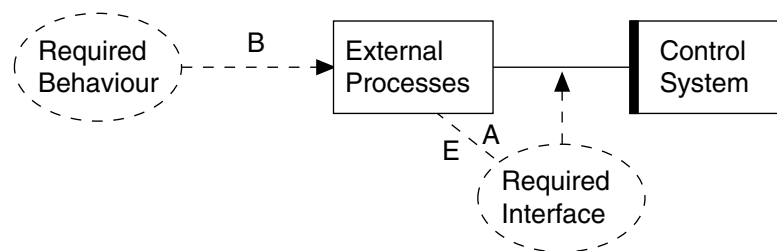


Figure 10: Process Control Frame

The *External Processes* domain is the only given domain. A new machine symbol has been introduced. The black side bar indicates that the *Control System* to be constructed is any physical machine with incorporated microprocessors. The *External Processes* domain, the *Required Behaviour* and the requirement phenomena *B* are the same as in the *Embedded Control Frame* of the previous section. But the interface of the machine is not defined yet. The *Required Interface* requirement states that building of the machine is to be based on peripheral devices that allow to detect and cause the external process phenomena *E* and *A* respectively.

The process phenomena *E* and *A* are directly related to the required functionality of the system. We therefore call these phenomena *essential specification phe-*

### 3 Problem Decomposition for Embedded Systems

---

*nomena*. Our goal is to base the solution of the functional control problem on the essential specification phenomena only.

We cannot expect that all of the essential specification phenomena are shared directly by the peripheral devices of the *Control System*.  $E$  and  $A$  are rather shared abstractly; they are used to relate the behaviour of the *External Processes* and the *Control System* in subject matter terms. For example, a clutch of a computer controlled vehicle is allowed to be closed (*Close*), when the angular velocity of the two clutch disks has become equal (*ReachEqualRPM*). The events *Close* and *ReachEqualRPM* are essential specification phenomena. However, the *ReachEqualRPM* event can be detected indirectly only by monitoring rotation pulses and comparing the measured rotation rates of the clutch disks.

#### *An Engineering Choice: Events and Actions*

For a given embedded system problem there are many choices of the set of essential specification phenomena possible. Because we aim to describe the functional solution by an event-driven reactive software machine, we constrain the sets  $E$  and  $A$  of essential specification phenomena to contain event phenomena only. This choice does not restrict the considered problem class because it is always possible to replace state phenomena by suitable event phenomena. We shall discuss the rules to be used to transform a problem formulation using shared states into one which is based on shares events later on. The requirement phenomena  $B$  are not restricted by this choice.

Thus  $E$  represents a set of event phenomena that are caused by the *External Processes* or by the physical time whereas  $A$  is a set of event phenomena that are caused by the *Control System*. In the CIP terminology used in the later chapters, the phenomena of the set  $E$  and  $A$  are termed *Events* and *Actions* respectively. We shall use the CIP notion already in the following discussion about the decomposition of the *Process Control Problem*.

#### *3.3.2 Process-Oriented Decomposition of the Process Control Problem*

We decompose the problem of building the *Control System* into two main problems. One main problem is to build a machine bringing about the required functionality of the *Control System*. This functional problem is solved by constructing a reactive machine specified by computer software. The other main problem is to connect this machine with the processes to be controlled. The connection problem involves both, the development and installation of suitable peripheral devices, and the development of the connection among peripheral devices and the reactive machine.

### 3.3 The Process-Oriented Problem Decomposition

Figure 11 represents graphically the structure of the problem decomposition down to the separated system and software engineering problems.

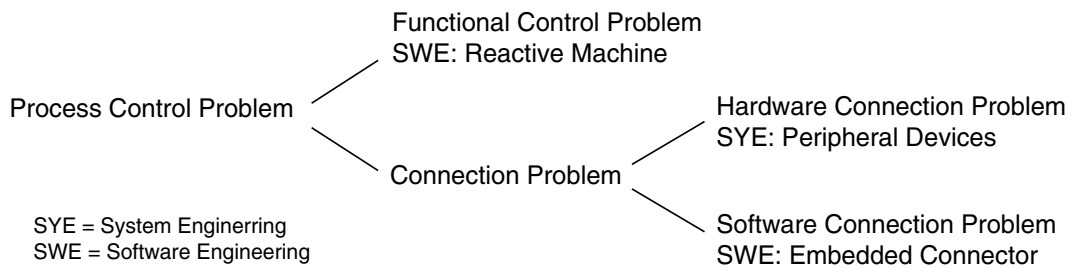


Figure 11: Decomposition of the Process Control Problem

The architectural picture in Figure 12 below shows the corresponding compositional structure of the resulting submachines. The two black side bars of the *Peripheral Devices* machine indicate these devices are hardware machines.

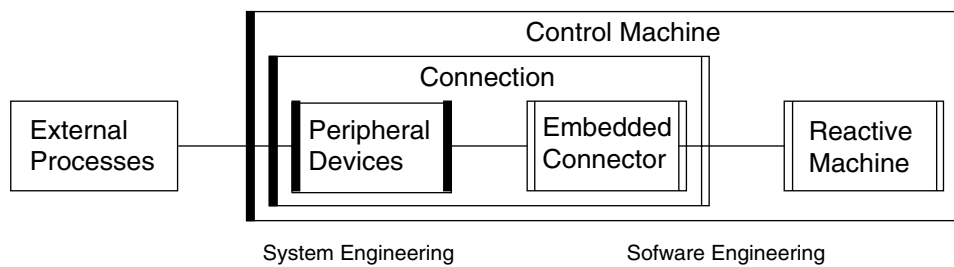


Figure 12: Compositional structure of the Control Machine

#### *A Remark Concerning a Domain Structure Clash*

Top level problems are usually decomposed in accordance with the involved engineering domains. A computer-based control problem, for instance, is decomposed into a system engineering and a software engineering problem. Our top level problem decomposition is different. It is guided by a process-oriented view focusing on the chief problem domain, the external processes to be controlled.

#### *3.3.3 Software Problems: Functionality and Software Connection*

The Functional Control Problem is formulated independently of the particular system engineering solution; that is, it is described independently of the installed peripheral devices. This is achieved by basing the functional solution on an abstract connection between the external processes and the reactive machine to be built. The connection problem consist of realising the abstract connection by installing a hardware connection by means of a particular set of

### 3 Problem Decomposition for Embedded Systems

peripheral devices, and by constructing a software connection to the reactive machine.

In the following we describe the problem frames for the software engineering problems resulting from the problem decomposition. The *Functional Control Frame* describes the problem of building the *Reactive Machine* while the *Soft Connection Frame* describes the problems of constructing the *Embedded Connector*.

#### *The Functional Control Frame*

In the problem frame of Figure 13, the machine to be constructed is the *Reactive Machine*. The dashed *Essential Connection* domain is a given abstract domain. The *Required Behaviour* must be brought about by the *Reactive Machine* through the *Essential Connection*.

An abstract domain describes the common properties of all valid potential realisations. In the final solution, an abstract domain must be realized by one or several concrete domains that satisfy the properties specified by the abstract domain. Thus the realisation of an abstract domain may represent a problem on its own. Our *Essential Connection* domain is to be realised by installing peripheral devices, and by developing a software connection that interconnects the peripheral devices and the *Reactive Machine*.

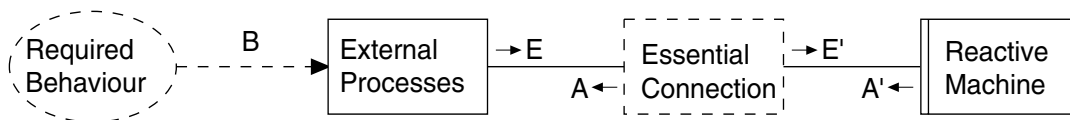


Figure 13: Essential Control Frame

The *Essential Connection* shares the essential specification phenomena  $E$  and  $A$  of the *External Processes*. This sharing is an abstraction too. It means, that a valid installation of peripheral devices must allow deducing all event occurrences of  $E$ , and causing all actions of  $A$ . The phenomena  $E'$  and  $A'$  shared with the *Reactive Machine* represent event notifications and action commands respectively. These interface phenomena of the *Reactive Machine* correspond one-to-one to the events  $E$  and actions  $A$  of the *External Processes* domain,

In addition, we suppose the *Essential Connection* to preserve the partial time order of occurring events and action commands. Events and action commands are in general not completely ordered in time because *External Processes* are concurrent, and because the *Reactive Machine* can consist of concurrent parts also. The stated behavioural property of the *Essential Connection* must be

### 3.3 The Process-Oriented Problem Decomposition

brought about when it is realised by means of the peripheral devices and the attached software connection.

The *Essential Connection* models asynchronous communication, that is, receiving and sending a particular message takes place at different points in time. This is important, because events and action commands can occur simultaneously at both ends of the connection. Even very short transmission delays can therefore not be neglected. We shall treat this serious real-time problem of race conditions in the section about context modelling.

#### *The Soft Connection Frame*

The software connection problem, represented in the problem frame of Figure 14, consists of completing the realisation of the *Essential Connection* when *Peripheral Devices* have been installed. The machine to be constructed is the *Embedded Connector* which interconnects the *Peripheral Devices* with the *Reactive Machine*. The *Peripheral Devices* and the *Reactive Machine* appear as given domains.

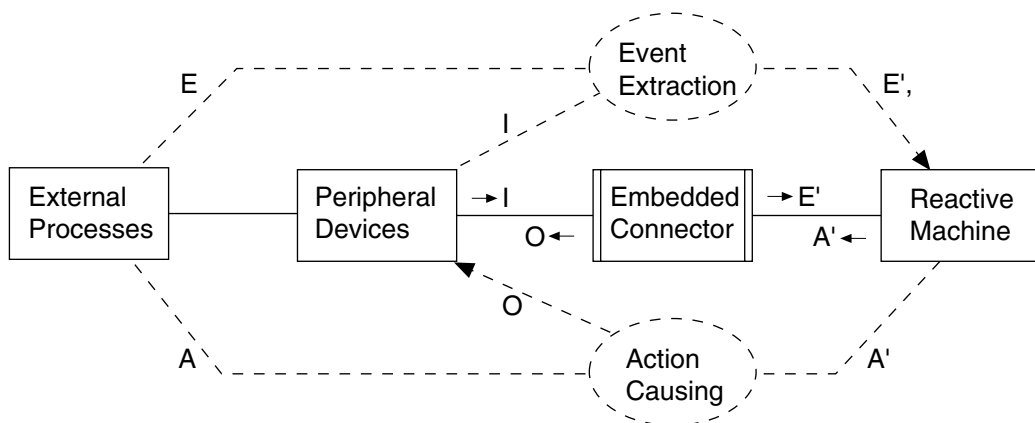


Figure 14: Soft Connection Frame

The *Embedded Connector* is connected at one side with the *Peripheral Devices* domain by the shared sensor and actuator phenomena  $I$  and  $O$ . On the other side, the *Embedded Connector* performs the event notifications  $E'$  for the *Reactive Machine* while the *Reactive Machine* issues the action commands  $A'$  for the *Embedded Connector*.

The *Event Extraction* requirement states that the *Embedded Connector* machine must interpret the sensor inputs  $I$  in order to recognise all occurrences of process event phenomena  $E$ . An extracted event is to be notified to the *Reactive Machine* by the corresponding event notification of  $E'$ .

### 3 Problem Decomposition for Embedded Systems

The *Action Causing* requirement states that the *Embedded Connector* has to perform the action commands  $A'$  received from the *Reactive Machine*. For each action command the *Embedded Connector* must initiate one or more instances of the shared actuator phenomena  $O$  to cause the commanded action.

Information of how a particular event can be extracted from  $I$ , or how a particular action can be caused by controlling  $O$ , is to be deduced from *Peripheral Devices* descriptions. A sensor specification describes for instance the physical interaction mechanisms on which the function of sensor is based.

#### *Sluice Control System*

We apply the problem decomposition on our sluice gate example.

Requirement states:

$$B = \{\text{Open, Shut}\}$$

Events and Actions:

$E = \{\text{ReachTop, ReachBottom}\}$  – the gate reaches the top or bottom position

$A = \{\text{Raise, Lower, Stop}\}$  – acts on the motor to raise, lower or stop the gate

States and events shared between peripheral devices and embedded machine:

$I = \{\text{top, bottom}\}$  – shared boolean interface variables

$O = \{\text{on, off, pos. neg}\}$  – shared pulses

And here is the solution:

#### **Reactive Machine**

The solution is given by a singular state machine:

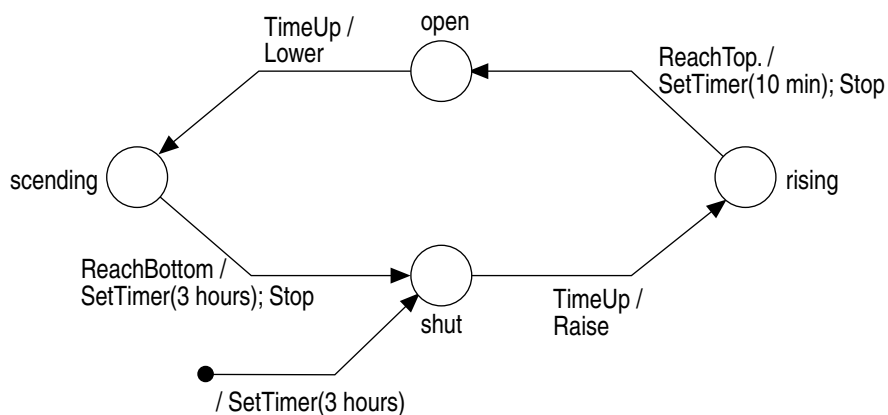


Figure 15: Finite state machine model of the reactive machine

#### Embedded Connector

The solution is a cyclic program that scans the sensor variables *top* and *bottom*, and triggers the reactive machine if an occurred gate event is recognised:

*top* changing to True -> event notification *ReachTop*

*bottom* changing to False -> event notification *ReachBottom*

In addition, a procedure implements the action commands issued by the reactive machine by initiating actuator pulses *on*, *off*, *neg* and *pos*:

*Raise* -> send the pulses *pos* and *on*

*Lower* -> send the pulses *neg* and *on*

*Stop* -> send the pulse *off*

The timer is also provided by the *Embedded Connector*. When the state machine sets a timer, the computer time is monitored to trigger eventually the state machine with a *TimeUp* event.

#### Conclusion

The problem of constructing an embedded system is decomposed into a functional and a software connection problem. These problems are considered as subproblems of the general computer-based system engineering problem. The development of the embedded functions is based on an abstract connection between external processes and embedded functions. The development of the software connection is considered as subproblem of implementing the abstract connection by means of peripheral devices and a software connection.

Viewing the software engineering problems (functionality and software connection) as subproblems of the general system engineering problem is not a purely academic task. In many real projects changes or replacements of peripheral devices arise during the software development phase; or worse, the devices are even not available at software development inception. In any case, the relation between system and software engineering activities play a central role for controlling the complete system development process.

