4 Process-Oriented Development of Embedded Functions

Domain-oriented software development emphasizes relationships between the problem domain and the formulated solution. Our domain-oriented decomposition of the embedded system problem bases the development of embedded functions on an abstract connection to the external processes, which represent the chief problem domain. We therefore introduce the term *process-oriented* development to denote domain-oriented development of embedded functions.

Process-orientation specialises the general principles of domain-orientation for the particular problem class of constructing embedded functions. We shall formulate two development concepts that describe how the description and the correspondence principle of domain-orientation are to be applied when we solve the functional problem in embedded system development.

In the first section of this chapter, we formulate the mentioned process-oriented development concepts. The two subsequent sections describe each of these concepts in detail.

4.1 The Process-Oriented Development Concepts

Two concepts describe how the principles of domain-orientation are applied in embedded system development. Both concepts base the development of embedded functions on the behaviour of the external processes. The first concept proposes to describe embedded functions by means of compositional behavioural models. The second concept claims to base embedded functions on behavioural correspondences with the external processes.

Description Concept: Compositional Behavioural Models

> Embedded functions are to be specified
> by means of an architectural model
> of interacting extended finite state machines.

The concept describes how the description principle of domain-orientation is applied to the development of embedded functions. The construction of the functional components is based on a reactive behavioural model called reactive machine, consisting in general of several concurrent parts. The reactive machine is a stimuli-driven abstract machine extended by computational primitives. The abstract machine is a mathematical model that defines the reactive behaviour of the functional components. Computational primitives, such as conditions and operations, are defined in a programming language; they are executed during reactions of the abstract machine.

The reactive machine is to be constructed by means of architectural composition. That is, a composite behavioural models is built by interconnecting components through connectors that mediate interaction. The compositional modelling approach ideally supports the construction process proposed by the correspondence concept.

Correspondence Concept: Behavioural Compatibility ascertained by a Context Model

> Embedded functions are to be based on a context model
> that establishes behavioural correspondences
> with the external processes.

The concept describes how the correspondence principle of domain-orientation is applied to embedded system development. The development of the reactive machine is based on a context model that establishes behavioural correspondences between reactive machine and external processes. A context model consists of independent state machines called context agents. The behaviour of a context agent must be compatible with the behaviour of the connected external process; that is, it must accept and produce valid sequences of input and output stimuli only. Invalid input stimuli are rejected as context errors.

The task of the initially developed context model is to maintain an ongoing behavioural relationship with the external processes. The required collective functional behaviour of the reactive machine is developed in a subsequent phase by introducing further state machines that coordinate the context agents.

4.2 Compositional Behavioural Models

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

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

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

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

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

In a classical paper, Harel and Pnueli [REF] have introduced the distinction between transformational and reactive systems to characterize the difference between common information systems and embedded real time-systems: "A transformational system accepts inputs, performs transformations on them and produces outputs. In contrast, reactive systems are repeatedly prompted by the outside world and their role is to continuously respond to external inputs." Transformational systems may ask for additional inputs from time to time and

produce outputs as they go along while reactive systems are supposed to maintain a certain ongoing relationship with its active environment.

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

Reactive Behaviour Described by a Finite State Machine.

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

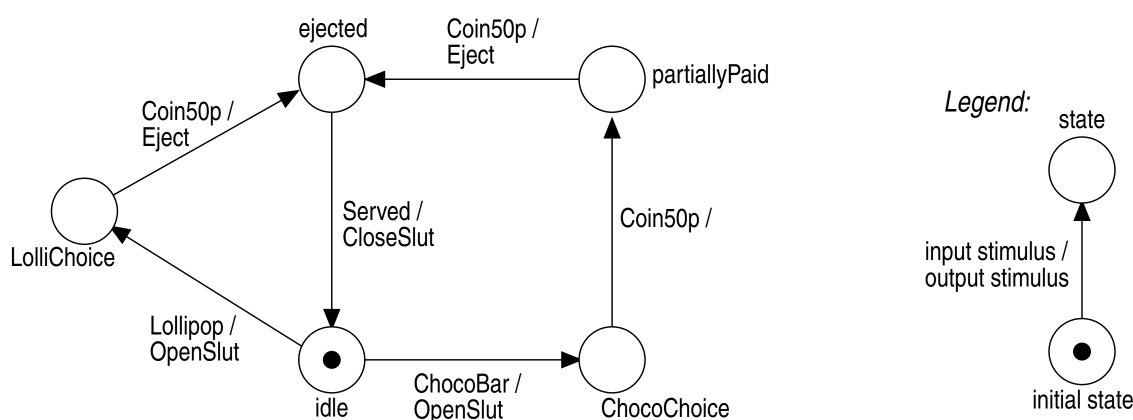
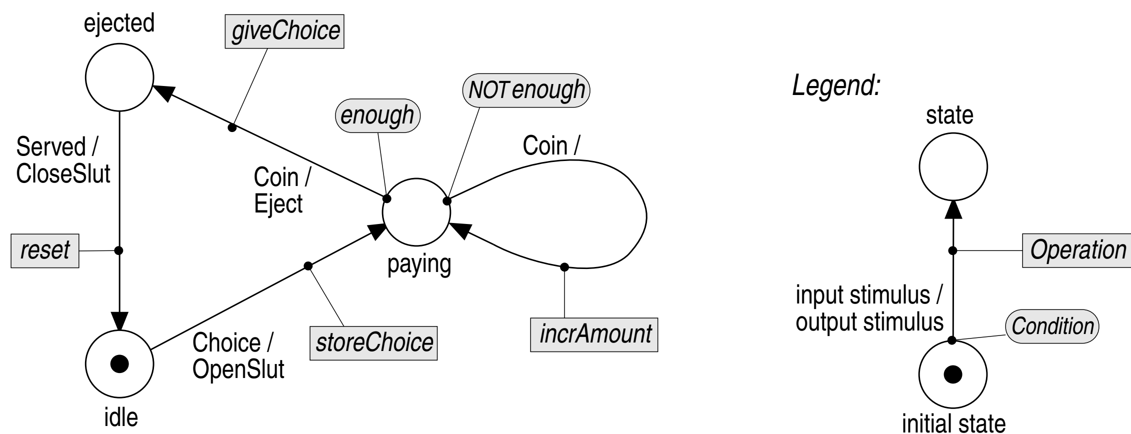


Figure 16: Finite state machine

The finite state machine depicted in Figure 16 controls a simple selling machine for lollipops and chocobars. The reaction on inserted 50p-coins depends on the previously pressed radio button for lollipops and chocobars.

Extended Finite State Machines.

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



Data Types
tProdNr DEF `struct{int prodNr;}`
tValue DEF `struct{int value;}`

Input Stimuli
Choice: tProdNr
Coin: tValue
Served

Output Stimuli
Eject: tProdNr
OpenSlut
CloseSlut

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

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

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

Figure 17: Extended finite state machine

The extended finite state machine of Figure 17 controls a selling machine that offers four different products. The prices of the individual products are stored in the statically initialized array variable *price*. The integer variables *productChoice* and *amount* are used to store the selected product and the amount of money inserted by coins. The input stimuli *Choice* and *Coin*, and the output stimulus *Eject* carry the data declared by the associated data type. The data are handled by the various operations associated to state transitions. The condition *enough* is used to resolve the non-deterministic behaviour in the state *paying*.

A state machine is deterministic if in every state at most one state transition can be triggered by a particular input stimulus. If a state machine is non-deterministic, the non-determinism can be resolved by exclusive guard conditions associated to state transitions. Such conditions can in general depend on the input data and local variables of the state machine. Although the basic plain state machine is non-deterministic, the extended state machine can resolve the ambiguity and represent a deterministic behavioural component. For example, the state machine of the selling machine in Figure 17 is non-deterministic in the state *paying* because two different state transition can be triggered by the *Coin* input stimulus. The condition *enough* used to resolve the non-determinism depends on the coin value, on the stored amount, and of the price of the selected product.

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

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

The qualitative and quantitative description of reactive behaviour supported by extended state machines have different characteristics. The qualitative description supported by the plain state machine model does in general not show

much regularity. It is typically the result of distinguishing a bounded number particular cases. The quantitative description is usually much more regular. Due to its algorithmic nature it can be based on computation rules, calculations and data processing functions.

4.2.2 Asynchronous and Synchronous Cooperation of State Machines

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

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

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

Asynchronous Cooperation.

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

from the buffer. Examples are the SDL channels or the well known FIFO message queues of real-time operating systems.

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

Synchronous Cooperation.

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

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

Combination of Asynchronous and Synchronous Cooperation

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

4.2.3 Architecture-Based Modelling of Interaction

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

Modelling of Interaction

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

The elementary components of our abstract machine are extended finite state machines. Each of these components represents an elementary state machine that is extended with computational primitives. By constructing interactions between the basic plain state machines we obtain the desired abstract reactive machine that models the global collective behaviour of the embedded system. In order to get a rigorously described abstract machine, interactions must be defined on the same level of abstraction as the plain state machines; for example by relating the output stimuli of one state machine to the input stimuli of an other state machine. In order to support an interaction that is related also to the computational extension of individual state machines, the modelled inter-

action is extended by mechanisms that allow the transport of data-items described in a programming language. The result are abstract reactive machines enhanced with computation power; that is, for the composed system we have a similar distinction of quantitative and qualitative reactive behaviour as for the elementary extend state machines.

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

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

Architecture-based Models of Interaction

The rigorous description of software architecture by means of architectural description languages (ADL) is a new emerging discipline of software engineering. Rigorous architectural system description separates the description of local behaviour and interaction. We shall adopt the concept of ADL's for the description of cooperating state machines.

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

ADL's base the construction of software systems on rigorous *architectural composition*: that is, the global system behaviour is defined by composing behavioural components by means of interaction connectors, as shown in Figure 18. Components represent behavioural artifacts that are inert with respect to composition. Connectors mediate interaction between components and define so

how the behaviour of the singular components combine to the global behaviour of the composed system.

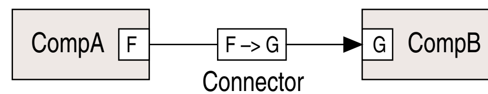


Figure 18: Interaction between two components

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

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

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

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

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

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

Problem Classes, Architectural Style and Modelling Frameworks

The kind of interaction to be chosen to solve a particular problem depends obviously on the class of the problem. The task of a domain-oriented method is to propose a suitable architectural style. An architectural style provides a set of connector types that support certain interaction mechanisms. A client-server style for instance is based on various types of service invocation mechanisms; working with pipes and filters is another architectural styles used to build information processing systems. The CIP modelling framework provides an architectural style supporting the domain-oriented construction of complex reactive systems.

A tool-based modelling framework based on a particular architectural style can be considered as a tool kit which places a suitable collection of modelling elements at the users's disposal. The main modelling activity consists of creating instances of components and connectors, and interconnecting them. Thus, instead of describing a model by expressions of a modelling language, models are constructed, like physical machines, by composing and connecting machine elements. Each modelling action initiated by the tool user causes an immediate update in the data model of the model under construction.

Such tool-based modelling frameworks not only allow transforming models into executable code, but even models under construction can by analysed automatically. CIP Tool, for example, allows the user to view all potential interaction sequences of a model under construction (see interaction analysis).

Interaction within Statecharts is implicit and global.

The widely used Statecharts models [REF] define interaction between orthogonal state machines by identifying input and output stimuli with global events of the modelled reactive machine. Each of the input and output stimuli of the orthogonal state machines that are equally named designate the same global event. The supported interaction mechanism is a global event broadcast. In addition, state transitions of a state machine may depend globally on the states of other state machines. The approach of modelling interaction by means of globally shared events and states has two major drawbacks. First, defining interaction mechanisms by means of global constructs violates the principles of modularity and information hiding. It is difficult to alter or reuse a part of such a model, because side effects due to changes in a particular state transition diagram must be expected in the whole model. Second, the effective interaction structure is described implicitly. To understand for example the dependencies of a particular event broadcast we must inspect the state transition structures of all orthogonal state machines.

4.3 Behavioural Compatibility ascertained by Context Models

The central task of an embedded system is to monitor and control the behaviour of a number of external processes. Domain-oriented system development is therefore based on behavioural relationships between the embedded system and the external processes. Such correspondences are obtained by building a reactive machine that embodies a collection of state machines termed context agents. Each context agent communicates with one of the external processes. The behaviour of a context agent must be compatible with the behaviour of the external process it is connected to. Behavioural compatibility as a strong behavioural relationship ensuring that the context agents can maintain an ongoing interaction with the active external processes.

The context model formed by independent context agents offers the basis for developing the global functionality of the embedded system. The required collective reactive behaviour of the full system is modelled by introducing further state machines that interact with the established context agents.

4.3.1 External Behavioural Properties and Valid Event Sequences

The design of control functions depends on both, the functional requirements and the intrinsic behavioural properties of the controlled processes. The intrinsic behaviour of the external processes is therefore a primary concern of embedded systems, and domain-oriented development is correspondingly based on the behavioural properties of the controlled processes.

Take for example a lift system. A simple behavioural property of the lift car is that the car must have left the first or third floor before it can reach the second floor. Thus the event *ReachSecondFloor* is always preceded by the event *LeaveFirstFloor* or *LeaveThirdFloor*. The possible event sequences cause corresponding sequences of event notifications at the reactive machine interface. We call these input stimuli sequences *valid* sequences, and each stimulus of such a sequence is called *valid* input stimulus.

When an external process is controlled by the reactive machine, it depends in addition on the caused actions what events can occur. In our lift example, the car leaves the first floor and reaches the second floor only when the lift motor turns in the appropriate direction. The events *LeaveFirstFloor* and *ReachSecondFloor* occur in this order if and only if the *MotorRaise* action has previously been caused by the reactive machine; the *MotorScend* action would lead to different events.

The dependency of event occurrences on caused actions is a main peculiarity of embedded systems. This particular problem character reflects the central purpose of control systems, namely to influence the behaviour of the monitored processes. Our goal is therefore to capture these behavioural dependencies by basing the development of the reactive machine on sequences of valid input stimuli.

4.3.2 Behavioural Compatibility

The purpose of behavioural correspondence is to base the development of embedded systems on behavioural structures that correspond to behavioural properties of the controlled processes.

As described in section 3.3, we build reactive machines that consist of concurrent clusters of synchronously cooperating state machines. Each cluster represents a sequential machine that is usually triggered by event notifications caused by different external processes. A valid input sequence for a cluster is therefore an interleaved merge of valid input stimuli sequences caused by the individual external processes. In order to disentangle these interleaved stimuli sequences we base the construction of a reactive machine on a set of state machines termed *context agents* that are connected each to one of the external processes. By interconnecting every external process with a corresponding state machine we define a static correspondence between behavioural entities of the environment and the embedded system (Figure 19).

External event sequences	Sequence of event notifications	Dispatched event notifications
ProcessA: A1, A2		AgentA: A1, A2
ProcessB: B1, B2, B3	B1, A1, B2, C1, B3, A2	AgentB: B1, B2, B3
ProcessC: C1		AgentC: C1

Figure 19: Static correspondence between external processes and context agents

In addition to the static process correspondence, we require a behavioural correspondence between each context agent and associated external process. This dynamic correspondence is based on the notion of *behavioural compatibility.* Behavioural compatibility is originally defined as a property of communicating state machines:

> The behaviour of two communicating state machines is compatible, if exactly the message sequences that can be produced by one state machine are accepted by the other one.

When we claim behavioural compatibility between a context agent and the corresponding external process, conditions on behavioural properties can be imposed on the context agent only:

> The behaviour of a context agent is compatible with the behaviour of the associated external process, if the context agent accepts exactly the input stimuli caused by the external process, and if the context agent produces output stimuli that cause valid actions only.

The stimuli sequences that a context agent can accept and produce are determined by its state transition relation, described by a state transition diagram. The development of a context agent's state transition relation must therefore be based on the intrinsic behavioural properties of the associated external process.

The set of context agents embodied in a reactive machine constitute a context model that ensures that the functional solution is based on valid input stimuli only. The full reactive machine is built in a subsequent development phase by introducing further state machines which interact and communicate with the context agents in order to bring about the required global behaviour of the embedded system.

Due to abnormal external process behaviour or event detection errors, we cannot exclude the occurrence of invalid input stimuli. Although a behavioural compatible context agent does not react on an invalid input by executing a state transition, an invalid stimulus must in general not be ignored. In the implementation of the reactive machine, a rejected invalid input is expected to throw an exception. The exception handler can for example generate a specific error event that is notified subsequently to the reactive machine. The triggered error agent can then induce a suitable change of the behaviour of the reactive machine.

Often we extend the state transition relation of a context agent so that it can react on invalid event notifications too. With such error extensions, invalid input stimuli become valid, although they are recognised as incorrect. The context agent can handle an incorrect input stimulus directly within the triggered state transition. By such extensions of the context model, error handling capabilities become a part of the modelled system functionality.

Quasi-Continuous Control.

Continuous external processes are monitored by periodic sampling events that trigger control algorithms executed by the embedded system. Such algorithms, integrated as state machine extensions, describe system reactions quantitatively. The development of control algorithms is also based on behavioural properties

of external processes, but these properties are typically described by means of differential equations. Corresponding notions of behavioural correspondence, such as criteria of control stability, are supported by classical control theory. Our notion of discrete behavioural compatibility becomes trivial because there is only a singular sampling event that is notified repeatedly to the controlling extended state machine.

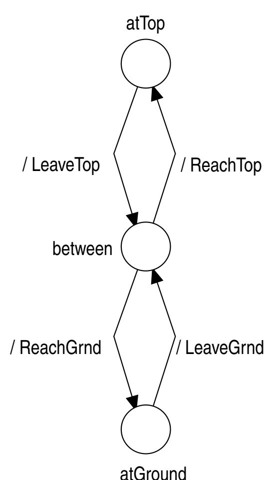
4.3.3 Behavioural Compatibility with Process Models

We illustrate context modelling by four examples. The first two examples are trivial with respect to behavioural compatibility; they show two cases where the context agent has the same behaviour as the external process. The third example demonstrates that behavioural compatibility can in general not be based on process models incorporated in the reactive machine. The fourth example illustrates the serious problem of external race conditions.

In all examples the external process is a simple lift with two floors. The relevant behaviour of the external process is always modelled by a state machine. Occurring process events are denoted as output stimuli of the process model while actions caused by the reactive machine appear as input stimuli. The behaviour of the connected context agent is described in the graphical CIP notation.

Example 1 – Monitoring the car position of a two-floor lift

External Process Model



Context Agent

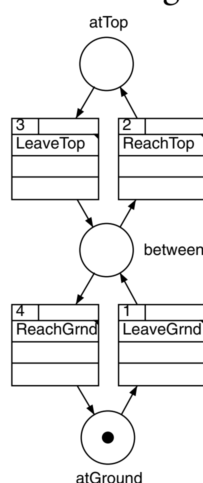


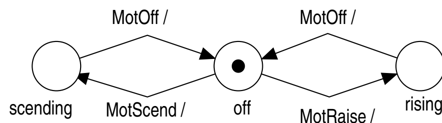
Figure 20: Sending external process model and receiving context agent

In the first example (Figure 20), the task of the system is to produce information about lift car travels; the lift is controlled manually. Establishing behavioural compatibility is trivial because the context agent can have an identical state transition structure as the process mode (l. Event notifications update

simply a copy of the external process model incorporated in the reactive machine.

Example 2 – Controlling the lift motor

External Process Model



Context Agent

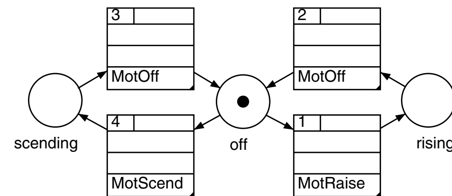


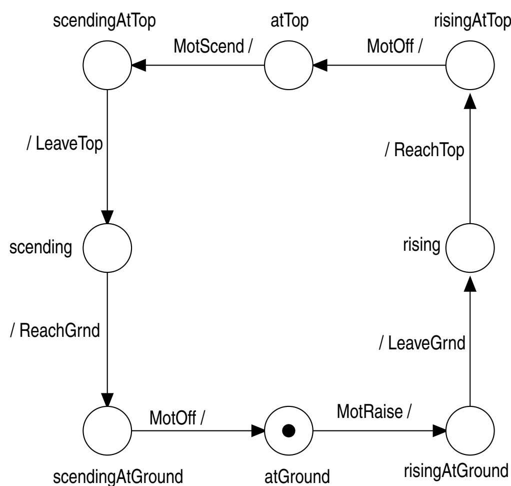
Figure 21: Receiving external process model and sending context agent

In the second example (Figure 21), the system controls the lift motor. The control is based on a predefined scheduling table. The context agent has again an identical state transition structure as the external process model. In contrast to the first example, the state transitions of the context agent anticipate the caused state transitions of the external process model.

Remark: Context agents specified during the context modelling phase are in general incomplete state machines. The context agent of the second example (Figure 21), for instance, has no input stimuli specified. The state transition structure is to be completed in a subsequent development phase when interaction with other state machines of the reactive machine is defined.

Example 3 – Controlling the lift car

External Process Model



Context Agent

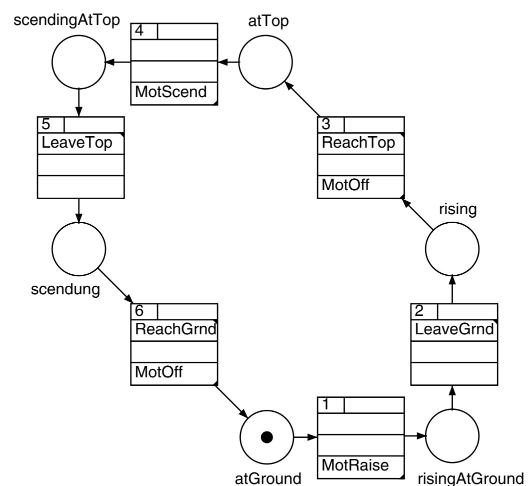


Figure 22: Bidirectional communication of external process and context agent

The third example in Figure 22 represent a usual lift control system; that is, the control of the lift is based on the monitored car position. The production of

action commands of the context agent depends on the behaviour of the external process. The necessary information about the behaviour is delivered by event notifications.

In the third example, it is not possible to use an identical state transition structures for the external process model and the connected context agent. With respect to event occurrences the reaction of the context agent is delayed while with respect to actions caused for the process model the context agent anticipates the behaviour of the process model. There exists for instance no state of the context agent that corresponds to the state *risingAtTop* of the process model. The state *risingAtTop* of the process model is necessary because the event *ReachTop* and the action *MotOff* occur, due to the asynchronous stimuli transmission, at different points in time. In contrast to the external process behaviour, the corresponding event notification *ReachTop* and the generation of the action command *MotOff* occur synchronously within the same state transition of the context agent. The temporal order of stimuli transmissions during the car travel from the ground to the top is viewed in the message sequence diagram of Figure 23.

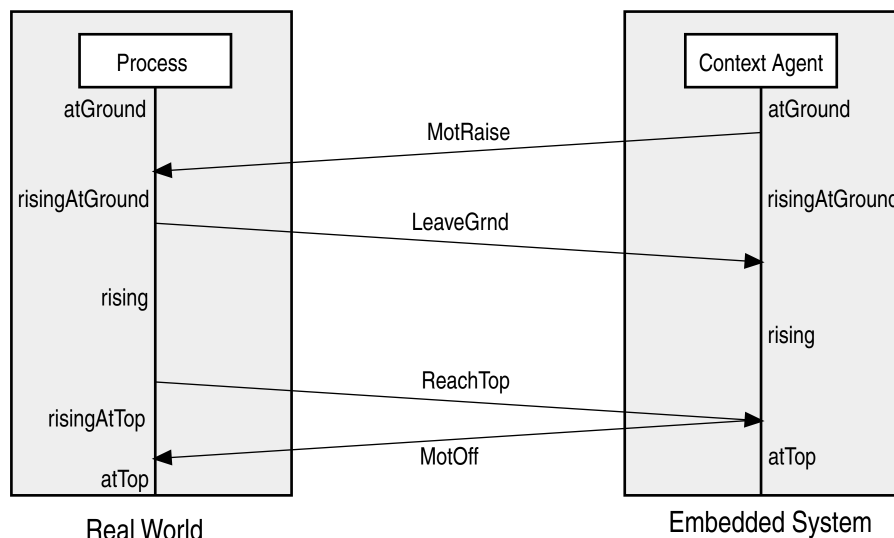


Figure 23: Message sequence diagram of a travel from the ground to the top

The compatible behaviour of the context agent is easy to verify in this example. The action commands are always produced at times where no further event notification must be expected. Thus the sequence of events and actions occurring in the real environment is equal to the sequence of event notifications and action command productions occurring at the context agent interface.

As long as action commands are produced when the external process is in an inactive state, or in response to an event notification of the controlled process,

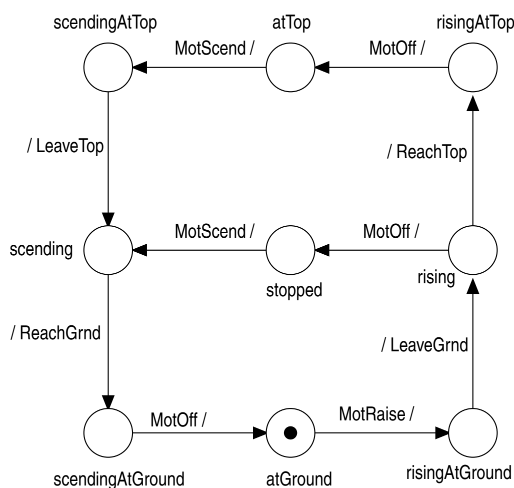
the event/action sequence and the corresponding input/output stimuli sequence at the reactive machine interface can in general be assumed to be equal. The assumption supposes that the latency time of stimuli transmission is much shorter then the minimal time duration between two subsequent event occurrences of the same external process.

Example 4 – Asynchronous stopping of the lift car

Context modelling becomes difficult when the reactive machine must act on a controlled process in response to events that occur asynchronously to that controlled process. Supposes for instance that our lift car must be stopped on its travel from the ground to the top as soon as a halt switch is pushed; when the switch is released the car must return to the ground.

The corresponding extension in the context agent and the external process model is depicted in Figure 24. We suppose that the lift car has no inertia; that is, the rising car stops immediately when the motor is turned off. The extended context agent can be triggered in the state *rising* by an other component of the reactive machine to cause the *MotOff* action. The corresponding action command is produced in the state transition that ends in the state *stopped*.

External Process Model



Context Agent

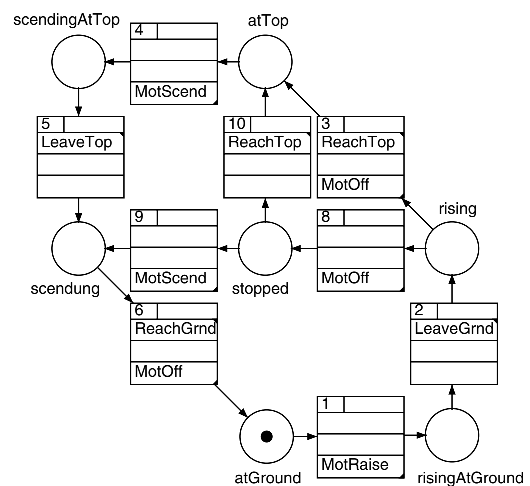


Figure 24: Bidirectional communication of external process and context agent

However, after stopping the motor we must expect during a short time interval that the event *ReachTop* is still notified. This race condition is due to the asynchronous stimuli transmission between external process and context agent. In our example it is possible that when the *MotOff* action is executed, the transmission of the event message *ReachTop* is taking already place.

The occurring bidirectional communication conflict appears in the message sequence diagram depicted in Figure 25 as crossed transmissions. The diagram also shows, how the time order of the *ReachTop* event and *MotOff* action in the environment and the time order of the corresponding stimuli occurrences at the reactive machine are intertwined.

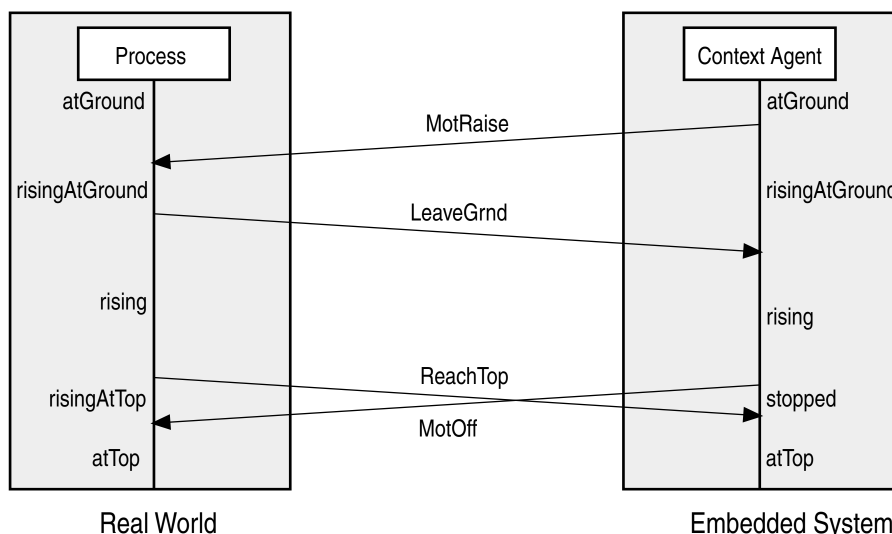


Figure 25: Message sequence diagram of a lately interrupted lift car travel

The problematic of asynchronous control of discrete event processes is serious because it is difficult to capture external race condition during the system test. Although external race conditions are the result of rarely occurring behavioural coincidences, the produced effect may be disastrous. In our example, a similar race condition would appear if we took the mechanical inertia of the lift car into account. However, the point of our example is to show that the problem of external race conditions is fundamentally related to the bidirectional asynchronous connection of external processes and embedded system.

Context modelling in the presence of external race conditions is treated in detail in chapter 5 where the development of CIP models is explained.

When Process Models are taken as Context Models.

The concept of basing a functional solution on a context model is not new. JSD (Jackson System Development) [REF] and some object-oriented real-time methods start the development with behavioural models of the monitored real world entities (process models). The real world models are then directly embodied as context model in the software system. Such context models represent in fact copies of the real world model and are updated whenever an event occurs. Using a real world model as context model to ensure behavioural compatibility works well for problems where the monitored real objects are not

controlled by the system. This is typically the case for business information systems. The functionality of such a system is typically to answer inquiries about the stored attributes of the monitored business objects.

Our lift examples three and four show, that the idea of incorporating external process models as context models in the software system cannot work for embedded systems, because the external processes are in general monitored and controlled as well. With respect to monitored events, the state of the context model is delayed, and with respect to produced actions it anticipates the state of the external process. Thus to capture the behaviour of controlled external processes we imperatively need to base our context model on context agents that have different behavioural structures than the models of the connected processes. In order to ensure an ongoing interaction with the external processes we require compatible behaviour of the context agents; that is, exactly the valid inputs stimuli must be expected and valid output stimuli must be generated only.