servicerobotik

autonome mobile Serviceroboter

Zentrum für angewandte Forschung
an Fachhochschulen

SmartSoft
The State Management of a Component

Christian Schlegel
Alex Lotz
Andreas Steck

Hochschule Ulm
University of Applied Sciences

Report 2011 / 01

Christian Schlegel, Alex Lotz und Andreas Steck
Hochschule Ulm
Prittwitzstrasse 10
89075 Ulm, Deutschland

schlegel@hs-ulm.de, lotz@hs-ulm.de, steck@hs-ulm.de
http://www.hs-ulm.de/schlegel

25. März 2011

SmartSoft
The State Management of a Component

Christian Schlegel
Alex Lotz
Andreas Steck

Contents

Chapter 1

Introduction

This document is the second technical report in the *ZAFH Technical Report Series.* The focus in this document is on the state management of a component. This domain is addressed by the state pattern in SmartSoft [1, chapter 5]. The original state pattern is described in [1, section 5.8]. The state pattern is extended by a generic state automaton. Its concept is described in the set of slides attached at the end of this document as appendix in chapter A. Some further descriptions are available in [3].

This document describes technical details from the implementation of the original state pattern in chapter 2 and the extensions of the state pattern for the generic state automaton in chapter 3. In addition, chapter 4 gives a practical example that demonstrates the usage of the new features in the state pattern on a source code basis.

The further development of SmartSoft and implementation of ACE/SmartSoft was funded by Robert Bosch GmbH (BOSCH) with the support of Dr. Michael Dorna.

Chapter 2

State Pattern in SmartSoft

This chapter describes technical details from the implementation of the *state pattern* as part of the whole SmartSoft idea [1, chapter 5]. Further details on the underlying ideas and motivation can be found in the set of slides attached as appendix in chapter A.

2.1 Master – Slave Relationship of the State Pattern

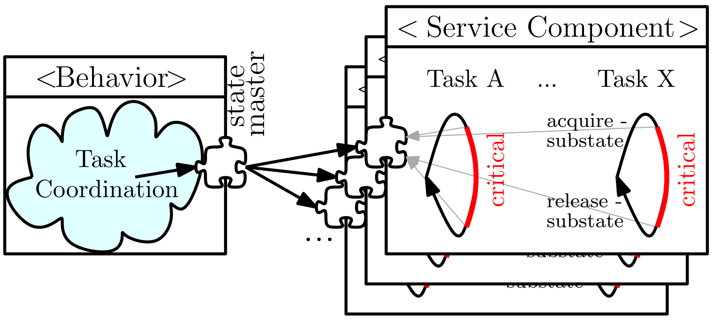


Figure 2.1: Master – slave relationship of the state pattern in SmartSoft
.

The *state pattern* [1, section 5.8] in SmartSoft supports a master-slave relationship to selectively activate and deactivate states. An activity can lock a state at the slave to inhibit state changes at critical sections as shown in figure 2.1. A critical section prevents an activity from being interrupted at an unsuitable point of execution. The state pattern gives the master precedence for state changes over the slave. As soon as a request for a state change is received from the master, the slave rejects locks for states that are not compatible to the pending state change of the master. The requested state change of the master is executed by the slave as soon as all locks for states affected by the state change are released. The state pattern is, for example, used by the task coordination component of the sequencing layer for graceful deactivation of component internal user activities.

2.2 Mainstates and Substates

A state slave defines a state automaton with mainstates on top level. Each mainstate is a mask for a subset of previously defined substates. A state master commands only mainstates. A task inside of a component acquires and releases only substates. This decouples the internal usage of the state slave (component's inner view) from the external state management (component's outer view).

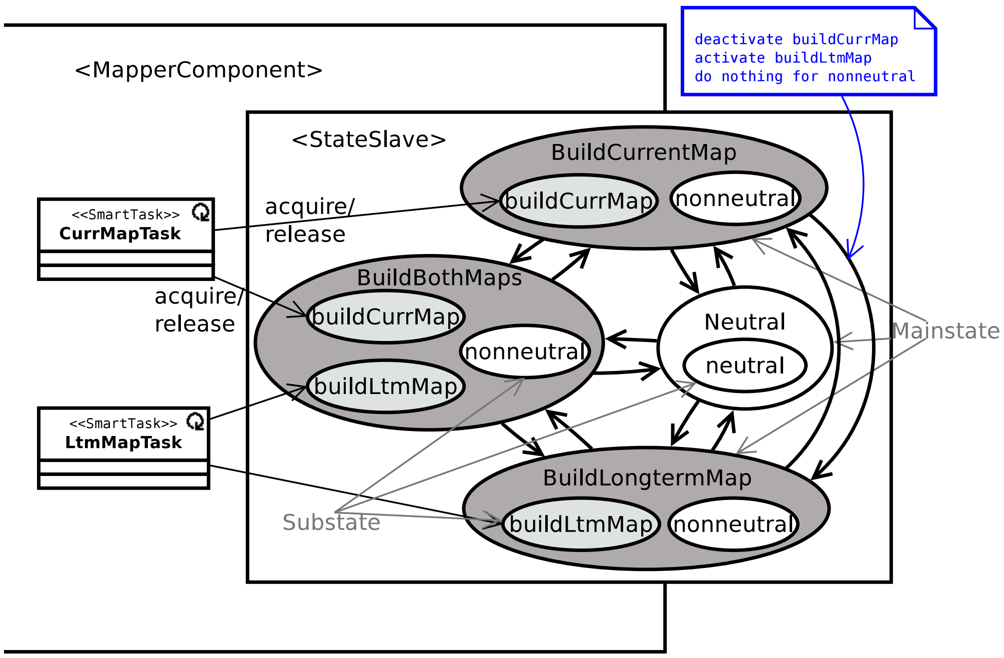


Figure 2.2: Mapper example in a state-slave port.

Figure 2.2 shows the `Mapper` component with a state slave service port. The implementation of this component consists of two tasks: the `CurrMapTask` and the `LtmMapTask` (other irrelevant details are left out for simplicity reasons). In general, a state slave automatically provides the mainstate `Neutral` with exactly one substate `neutral`. In addition, several mainstates and substates can be individually defined by a user in a state slave. Each user-defined mainstate automatically includes the substate `nonneutral`. This allows to start activities as soon as the mainstate `Neutral` is left and to stop them as soon as it is entered. In the `Mapper` component the mainstates `BuildCurrentMap`, `BuildLongtermMap` and `BuildBothMaps` are defined additionally to the `Neutral` mainstate. Further, an arbitrary combination of previously defined substates (apart from the substates `neutral` and `nonneutral`) can be attached to each user-defined mainstate. In the `Mapper` component, two user-defined substates are defined, the `buildCurrMap` and `buildLtmMap`. For example, if the mainstate `BuildCurrentMap` is currently active, the substates `nonneutral` and `buildCurrMap` are active as well. Thus, the `CurrMapTask` is able to acquire the substate `buildCurrMap` and enters its critical region, whereas the `LtmMapTask` is not allowed to enter its critical region and blocks on the `acquire` call. In case the mainstate `Neutral` is selected both tasks are blocked. This mainstate is particularly valuable to stop all critical activities of a component and to reconfigure its parameters or rewire its service ports.

The state pattern additionally provides the specialized mainstate `Deactivated`. In fact, this is a pseudostate which is used to command a state slave to switch into its `Neutral` mainstate as fast as possible. Thereby, all blocking calls caused by communication pattern usage inside of the component are instantly unblocked with a corresponding status code. Thus, each task is able to leave its critical region and release the corresponding substate. This is a powerful feature of the state pattern to enforce rapid deactivation of components in cases where a blocking wait is unsuitable for a scenario.

In case of the state pattern, the state master directly commands the subsequent mainstate in contrast to a regular state automaton where the events trigger certain state changes. Each previously defined mainstate can be chosen as a subsequent mainstate independently of the previous mainstate. This enables the state master port to set a component into each desired mainstate in a direct and simple way. At each point in time only one mainstate and its included substates are active. During a state-change all substates that are not included in the subsequent mainstate are deactivated and all substates that are new in the subsequent mainstate are activated. Substates which are available in both mainstates (the current and the subsequent mainstate) are not affected by the state-change and execute without interruptions.

In the `Mapper` example in Figure 2.2, a state-change from mainstate `BuildCurrentMap` to `BuildLongtermMap` deactivates the substate `buildCurrMap`, activates the substate `buildLtmMap` and does not interrupt the `nonneutral` substate.

A task that acquires a substate (which has to be deactivated due to a state-change) is blocked as soon as it tries to enter its critical region again (in its next run). This is independent of other tasks which might still hold the same substate. Thus, the substate can be deactivated as soon as the slowest of these tasks leaves its critical region (releasing the corresponding substate).

2.3 Implementation overview

The *state pattern* is implemented on top of the *query pattern* in SmartSoft (see class diagram in figure 2.3). Thereby, the *state master* internally uses the *query client* port to send state-change-requests to the *state slave*. The *state slave* pattern internally uses a *query server* to receive state-change-requests from the state master. All incoming state-change-requests are processed in an internal task of the state slave and an answer is replied back to the state master after a state-change has been successfully performed. In case of an illegal state-change-request, an answer with a corresponding status code is replied to the state master.

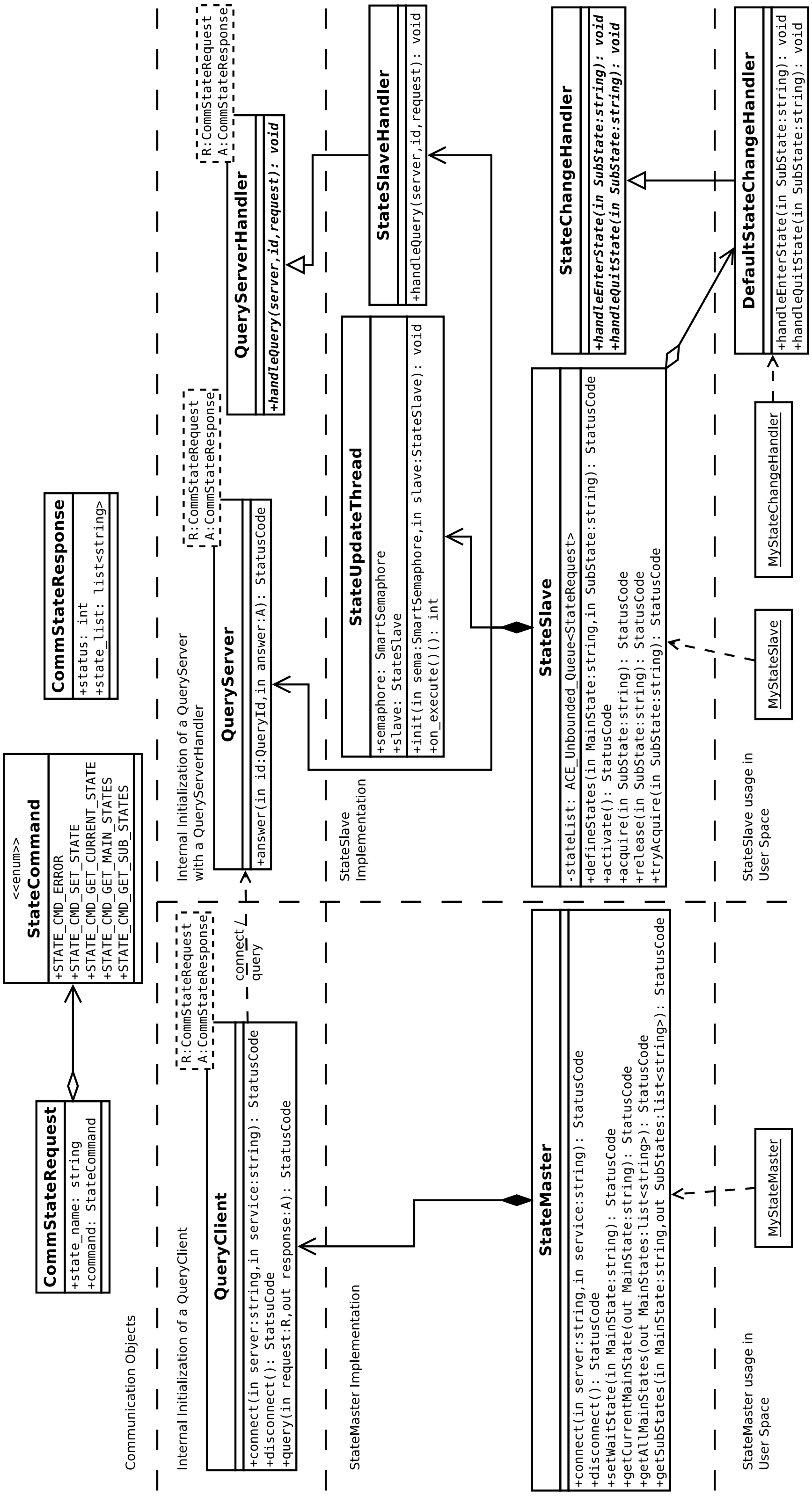


Figure 2.3: Class diagram for the state pattern (including internal query pattern).

2.4 Performing a state-change-request

The *state slave* comprises two main parts (see figure 2.3), an internal FIFO queue (the `stateList`) to store all state-change-requests from the state master and an internal task (the `StateUpdateThread`) which consecutively processes all these state-change-requests.

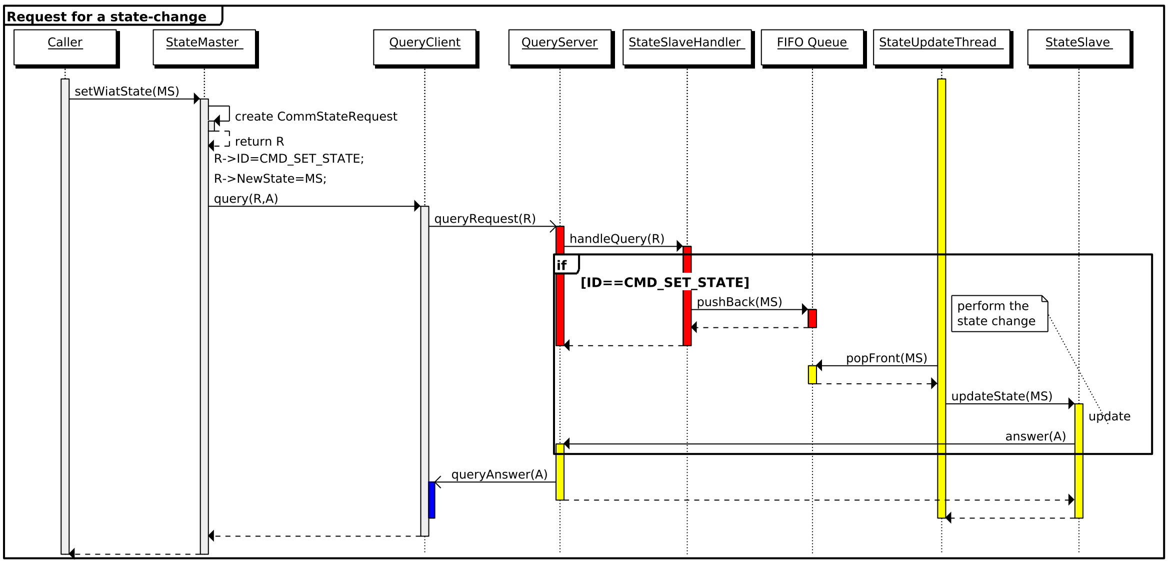


Figure 2.4: Sequence diagram for a state-change-request.

A state-change-request (caused by a call of the `setWaitState` method in `StateMaster`) is shown in the sequence diagram in figure 2.4. First, the `StateMaster` creates a `CommStateRequest` communication object with the command id `CMD_SET_STATE` and the name of the target mainstate `MS`. After that, the `query` command of its internal `QueryClient` port is called. The `QueryClient` internally calls a `queryRequest` and waits on the `queryAnswer` from the remote `QueryServer`. The `QueryServer` forwards the request to the registered `StateSlaveHandler` (derived from the `QueryServerHandler`), which in turn evaluates the command id and in case of `CMD_SET_STATE` pushes the target mainstate `MS` onto the FIFO queue.

The task `StateUpdateThread` blocks if the queue is empty or pops the top element from the queue otherwise. In the latter case the task calls the `updateState` method (from the `StateSlave`), which internally calls the `update` method and performs the real state-change-procedure. After all substates that are not included in the target mainstate are deactivated and all substates that are new in the target mainstate are activated, the `StateSlave` replies an answer to its `QueryServer` with a corresponding status code. The `QueryServer` sends a `queryAnswer` back to the `QueryClient` which releases the initial waiting thread of the `Actor`.

It is important to notice that an internal task is strictly necessary because the following reasons: First, the internal query server is used to command a certain mainstate as well as to request for the current mainstate. With a passive query server a request for the current mainstate would block in case of a currently pending state-change-request. Second, multiple state-change-requests are stored in the FIFO queue and are consecutively executed in the correct order. Finally, with a queue it is possible to prioritise certain state-change-requests like the command `Deactivated` for example as shown in the following.

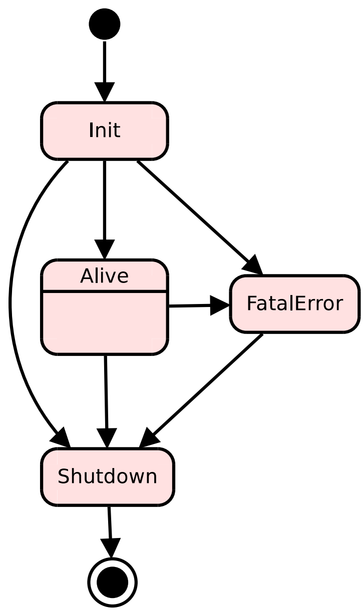
In the special case when the pseudo mainstate `Deactivated` is commanded, the state-change procedure is extended by a further step. After the `StateUpdateThread` popped the top element from the FIFO queue (see figure 2.4), the name of the mainstate `MS` is compared with the `Deactivated` keyword. In case of a match, the `StateUpdateThread` internally calls `blocking(false)` in its owner class `SmartComponent`. Thus, all blocking calls in this component caused by communication patterns (which wait on pending requests) are temporarily unblocked to enable all corresponding tasks to leave their critical regions. After all relevant tasks have left their critical regions, the `StateUpdateThread` internally calls `blocking(true)` to restore normal behavior in all communication patterns. After that, the state-change into the `Neutral` mainstate is completed and an answer to the `StateMaster` is replied.

Chapter 3

Generic State Automaton based on State Pattern

The concept for a generic state automaton based on the state pattern in SmartSoft [1, chapter 5] is originally described in the set of slides attached as appendix in chapter A. Some further details are available in [3]. This chapter describes technical details from the implementation of the generic state automaton based on the state pattern in SmartSoft.

3.1 Lifecycle of a component



Each component in a system goes through a set of standardized states during its lifetime (see figure on the right). At startup a component is in its `Init` state where all component's internal resources are initialized. If the component is fully initialized and is ready to deliver proper service, the component traverses into the `Alive` state. This is a regular state where a component executes its specific task. Further, a component can be commanded to shut down independent of the previous state. This can be done either from within the component itself (i.e. by firing a `SIGINT` signal) or from the outside of the component by commanding the `Shutdown` state-change. Both cases result in the same behavior to traverse into the `Shutdown` state, to clean up component's resources and finally to shut the component down. During initialization and later at runtime, critical errors can occur in a component. In this case the component traverses into the `FatalError` state. This state means that the component is not able to continue its service anymore and requires help from outside.

The lifecycle state automaton of a component defines generic modes for a component and a precise semantics for each of these states with their transitions. This allows to automatically supervise and orchestrate components at runtime.

3.2 Integration of a component's lifecycle into the state pattern

The lifecycle state automaton is independent of any robotic middleware or framework. The state pattern in SmartSoft provides suitable structures to easily integrate the lifecycle state automaton.

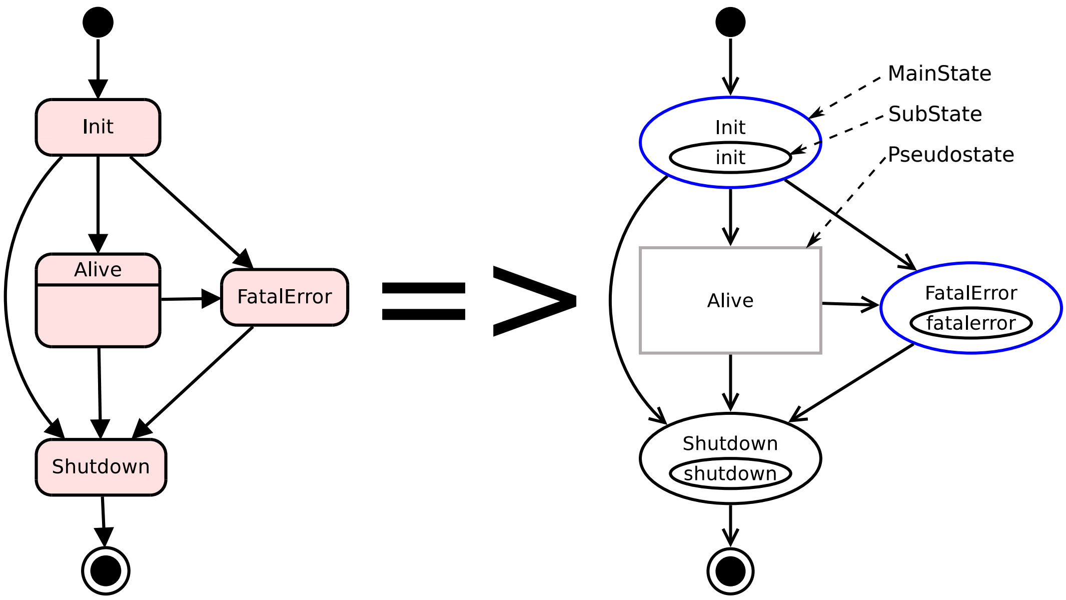


Figure 3.1: Integration of the component's lifecycle into the state pattern.

Figure 3.1 shows a representation of the lifecycle state automaton by the means of the state pattern. First, the lifecycle states are implemented as predefined mainstates in a state slave which are available from the beginning and are stable at runtime. In particular the mainstates `Init`, `Shutdown` and `FatalError` are created with exactly one substate. As a convention each mainstate name begins with a capital letter and each substate name begins with a small letter. Thus, each substate in the corresponding mainstate from the generic state automaton is of the same name (besides the first letter). This enables a component to use tasks which are active during the initialization of a component to manage its initialization procedure. In case of a fatal error, specialized tasks can be defined which execute suitable actions. Finally, during a shutdown of a component, special tasks can manage ordered clean up of resources.

The state **Alive** has a different semantics. This state, is at first, a pseudo state very similar to the **Deactivated** command in the original state pattern. In fact, the **Alive** state is a command which is used to transfer a component into its regular execution mode after all necessary resources in this component are fully initialized.

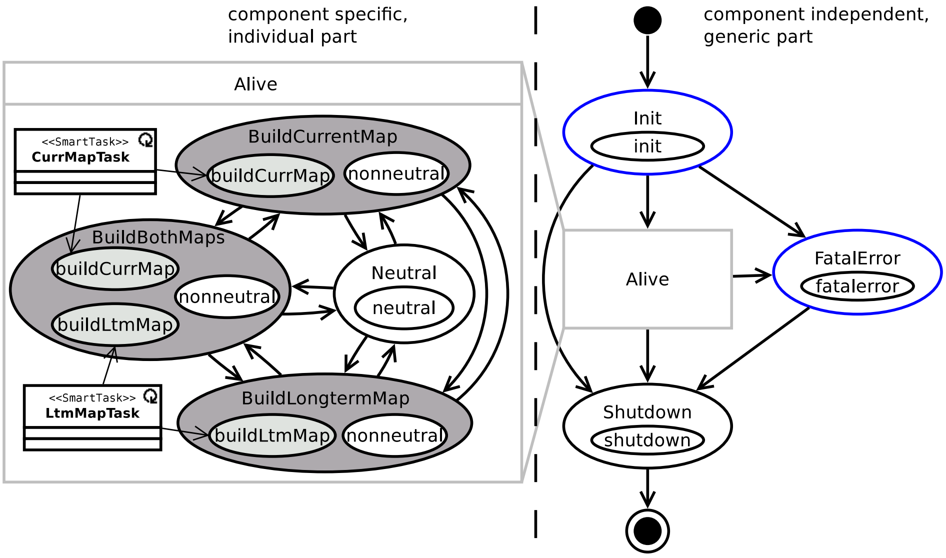


Figure 3.2: Mapper Example for the State Pattern

The original state pattern in SmartSoft consists of a component specific state automaton. In the extended state pattern, this state automaton is placed inside of the pseudo state **Alive** (see mapper example in figure 3.2). Thus, both the generic and the individual automatons are combined. If a component is in its **Alive** state, the component's individual state automaton is used with the exactly same semantics and behavior as it is in the original state pattern. In this case, all customized mainstates (and the mainstate **Neutral**) are externally visible and controllable from a state master. If a component is in process of initialization, shutdown or in a fatal error, transitions are performed according to table 3.1 (this table sums up all transition conditions which are originally defined in the set of slides attached as appendix in chapter A).

At startup a component is automatically set into the **Init** mainstate as soon as the state slave port is initialized. The transition from **Init** to **Alive** is only allowed from within the component itself, because only inside of the component all information are available to decide when all necessary resources are fully initialized. Outside of the component, the **Init** mainstate is visible in a state master and can be used to wait till the component is ready to run. During initialization a fatal error can occur inside of the component (i.e. a hardware part failed to initialize completely). The decision for this case is the local responsibility inside of the component and thus the transition is only allowed from within the component. Outside

	Target state					
Current state	Init	*Alive*	Neutral	User-defined	FatalError	Shutdown
Init	-/-	int./-	-/-	-/-	int./-	int./ext.
Alive	-/-	int./-	*predef.*/-	*predef.*/-	int./-	int./ext.
Neutral	-/-	-/-	-/ext.	-/ext.	int./-	int./ext.
User-defined	-/-	-/-	-/ext.	-/ext.	int./-	int./ext.
FatalError	-/-	-/-	-/-	-/-	int./-	int./ext.
Shutdown	-/-	-/-	-/-	-/-	-/-	int./ext.

int.	Transition is triggered *internally* (from state slave interface)
ext.	Transition is triggered *externally* (from state master interface)
predef.	Transition is *predefined* by the `setUpInitialState` method
-	Transition is *not* allowed

Table 3.1: Allowed transitions between different mainstates.

of the component, the `FatalError` mainstate is visible in a state master. This allows to react on this situation in a suitable way (for example this component can be commanded to shut down). Finally, during initialization of a component it might be necessary to shut the component down either from within the component itself (i.e. due to a local SIGINT signal) or commanded from a state master (i.e. because the initialization procedure took too long for the current situation in a scenario).

The command to switch into the `Alive` pseudo state stops all initialization activities and activates the initial mainstate of the component specific state automaton. Per default the initial mainstate is `Neutral`. In addition, during initialization of a component the initial mainstate can be changed to one of the customized mainstates by using the method `setUp-InitialState` of the state slave. From now on all customized mainstates and the `Neutral` mainstate can be externally orchestrated by a state master. During this regular execution, a fatal error can occur which is not solvable by regular error handling strategies inside of the component and which prevents the component from providing proper service. In this case the component is able to switch into the `FatalError` mainstate, which deactivates all activities inside of the component. This is useful, because the component does not simply disappear from the system but switches into a consistent mode and signals a problem which might be solvable on a higher level (the system level). During the execution it might be additionally necessary to shut down a component, either commanded from within the component (again, due to a local SIGINT signal), or commanded from the outside by a state master (i.e. because the component is not needed in the scenario anymore).

The `FatalError` mainstate is not used for regular error handling strategies inside a component. Moreover, all problems which can be solved locally inside a component should be solved locally and not delegated to the outside, because otherwise this leads to tightly coupled components with unclear responsibilities. Thus, the only way out of the `FatalError` mainstate is to shutdown the component (again, commanded either from within the component or from the outside by a state master).

Finally, a component in the `Shutdown` mainstate stops all other activities in the component and activates the shutdown procedure. Thus, all relevant resources (like hardware drivers) can be cleaned up and the component can be stopped in a coordinated way.

3.3 Implementation details of the state pattern extensions

The implementation of the state pattern extensions affects only the internal handling of states inside of the state slave communication port. Neither the underlying communication mechanism must be modified, nor the public interface of the state master must be changed. The interface of the state slave is extended by three additional methods (see figure 3.3). This allows to use the new state pattern in already implemented components without major modifications and to use all new features in ported or new components.

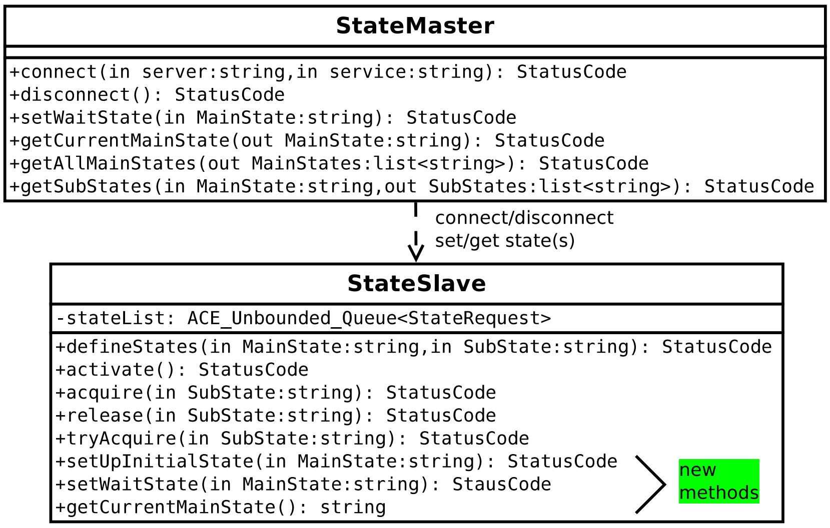


Figure 3.3: Extended interface of the State Pattern.

In the state master the two methods `getAllMainStates` and `getSubStates` behave exactly the same as before: they just return the mainstates and substates of the user-defined state automaton. The generic mainstates and substates from the lifecycle state automaton are not visible in these methods. They are implicitly known, because the are the same for all components. The method `getCurrentMainState` on the other hand returns the real current mainstate, even if it is one of the generic lifecycle states. The method `setWaitState` also behaves exactly the same as before. However, in addition the mainstate `Shutdown` can be commanded to trigger a remote component to shut down. Thus, the extended state master can be used either in the original way or can additionally be used with the new features.

The state slave communication port keeps its original interface and is extended by three new methods (see figure 3.3). The method `defineStates` allows to define any combination of mainstates with substates as before except the reserved mainstates and substates from the generic lifecycle state automaton. The methods `acquire`, `release` and `tryAcquire` can now additionally use the substates `init`, `fatalError` and `shutdown` to implement specialized tasks for the initialization and shutdown procedures inside of a component or tasks which execute appropriate activities in case of a fatal error.

The first new method `setUpInitialState` in the state slave can be used during initialization in a component to predefine the initial customized mainstate which is automatically activated as soon as the state slave is commanded for the first time to switch into the `Alive` mainstate. If this method is not used during initialization of a component, the default initial mainstate is automatically set to `Neutral`, which exactly resembles original behavior of the state slave.

The second new method `setWaitState` in the state slave is used to internally command the generic mainstates from the lifecycle state automaton. This is necessary to give a developer the freedom to specify when the initialization procedure is over, when a critical error occurs or when to shut down the componen. This method internally ensures correct transitions as defined in table 3.1, taking the previously active mainstate into account.

The third new method `getCurrentMainState` in the state slave is a helper method to ask for the currently active mainstate from within a component. This is particularly useful for observation and documentation purposes (like logging or runtime monitoring). The behavior of the method is to return the currently active mainstate or in case of a currently pending state-change to return the new mainstate to be activated. The latter is important to get correct information if this method is used within the state-change handler of the state slave.

3.4 Application of the new features of the state pattern

The new features in the state pattern support a component developer to design clear structures and to clearly separate responsibilities inside of a component. Thus, the developer is encouraged to strictly distinguish between activities, that are responsible to initialize component's internal resources and activities to clean up these resources. Also, the regular execution and the fatal error cases can be now simply separated in the internal implementation in a component.

The initialization and shutdown procedures without the usage of the state pattern are already described in the ZAFH Technical Report 2010/01 [2, chapter 3]. The same procedures are also possible with the new features in the state pattern, but now with a strict separation of concerns and structures. The initialization procedure of a component with a state slave is illustrated in the sequence diagram in figure 3.4. From a developer's point of view, additionally to the initialisation of a `SmartComponent` class, the `StateSlave` must also be initialized.

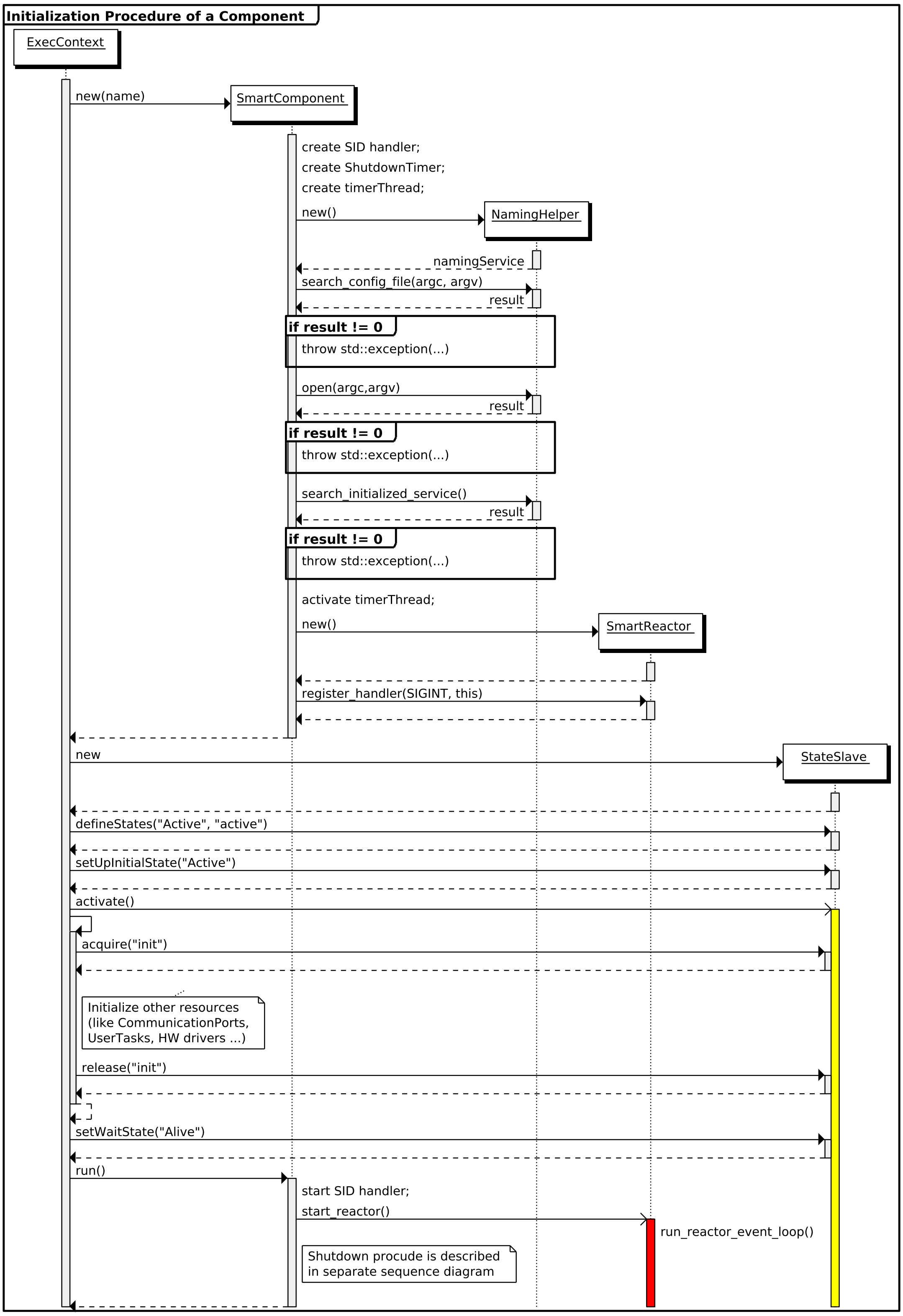


Figure 3.4: Sequence Diagramm for an Initialization Procedure in a Component.

It is the responsibility of a developer to use or not to use the state pattern. However, without the state pattern, the internal lifecycle state automaton is not explicated in the component and it is not possible to use them at runtime. The first important part during initialization of the `StateSlave` is the definition of the customized mainstates with corresponding substates. The simplest state automaton consists of the mainstate `Active` with the substate `active` (as shown in figure 3.4). After that, the initial mainstate can be defined by the method `setUpInitialState`. Again, if this method is not used, the mainstate `Neutral` is set per default as the initial mainstate. Next, the `StateSlave` can be activated to fully manage all states. From now on, the `StateSlave` is fully initialized and can be used by a remote state master to request and command mainstates. Inside of the component the substate `init` can be acquired by an arbitrary number of tasks to coordinate the component's individual initialization of resources. At the same time, the main thread can call `setWaitState("Alive")` to switch the component into regular execution mode. This call can be performed independently of the current progress in the initialization procedure. The main thread is simply blocked as long as the initialization tasks are not finished, and is thus unblocked as soon as the last initialization thread releases the substate `init`. Finally, a call of the `run` method starts the infinite loop for the internal event handling (in `SmartReactor`).

Further, one or several tasks can be defined which internally acquire the substate `fatalError`. These tasks are idle as long as the component is not in the fatal error state. In the regular case these tasks are even never activated, if no errors occur during the whole lifetime of a component. However, if a fatal error occurs individual actions for each component can be defined. Such an action is, for example, to trigger a higher level (system level) error handling routine, or to inform the task coordination component about the error.

After a component has successfully switched into the `Alive` mainstate, the component's individual state automaton is used for state management. In the simplest case, this state automaton consists of the mainstates `Neutral` (with the substate `neutral`) and `Active` (with substates `nonneutral` and `active`). Again, the mainstate `Neutral` is used to deactivate all component's internal activities and thus to save resources and to be able to reconfigure this component without the risk of interrupting critical activities at unsuitable points of execution. On the other hand, the mainstate `Active` can be used to activate all internal activities (tasks) in a component which calculate the data for service ports of this component. Other individual state automatons with several different activity modes are also possible as demonstrated with the mapper example (see figure 3.2).

Finally, the mainstate `Shutdown` is used to clean up component's resources before the component shuts down. This procedure is shown in the sequence diagram in figure 3.5. As already shown in figure 3.4, the component starts its internal execution by calling its `run` method. The shutdown procedure now can be triggered either from within the component itself (by catching the SIGINT signal inside the `handle_signal` method in `SmartComponent`), or com-

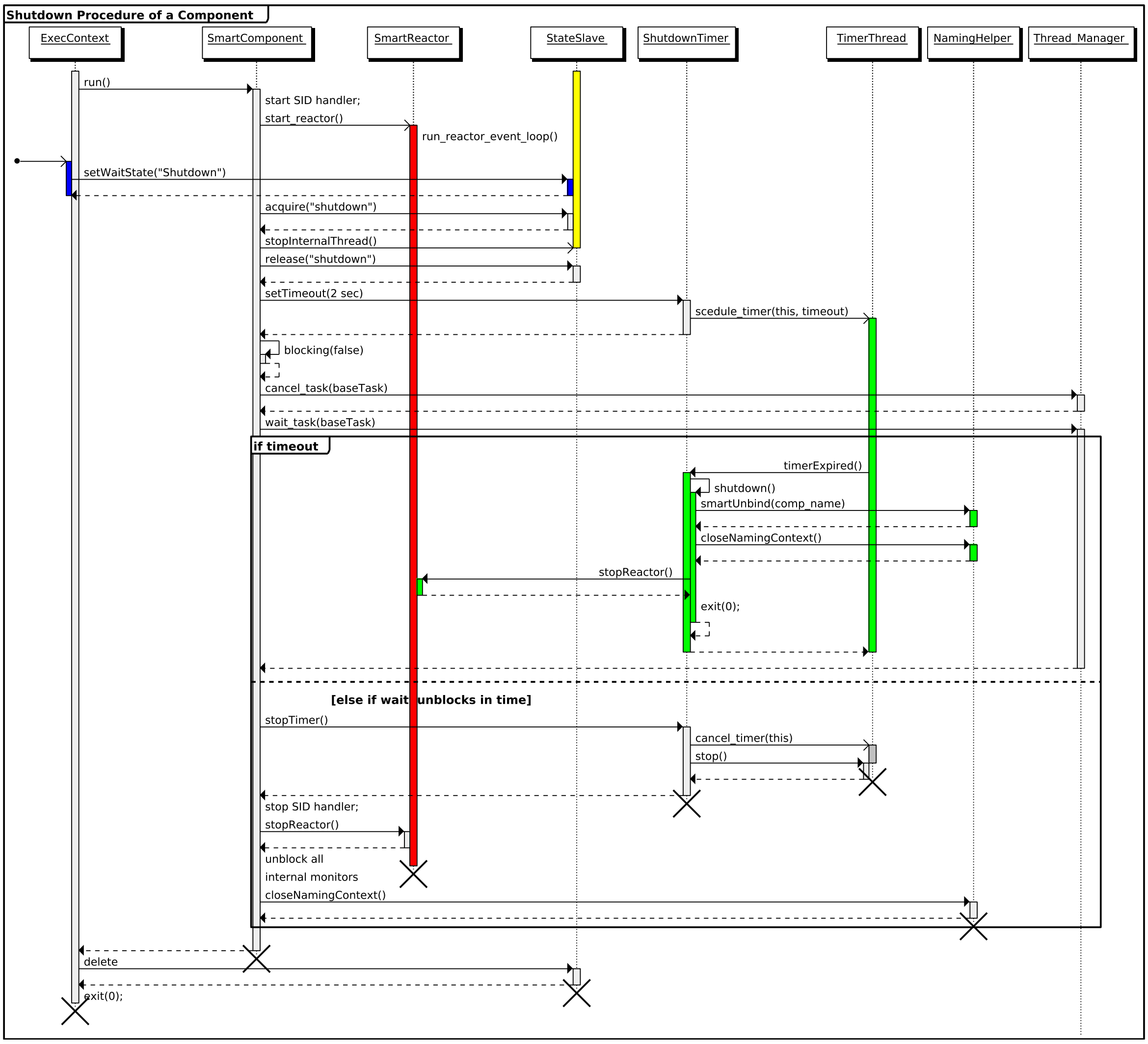


Figure 3.5: Sequence Diagramm for a Shutdown Procedure in a Component

manded by a state master from the outside the component. In both cases, the `StateSlave` is commanded to switch into the mainstate `Shutdown`. Thus, the same behavior is implemented independently of where the shutdown command is triggered from. After, the `StateSlave` has activated the mainstate `Shutdown`, the method `run` is able to acquire the substate `shutdown` and thus the corresponding thread is unblocked. Since the mainstate `Shutdown` is always the very last mainstate in a component before the component goes down, the internal thread inside of the `StateSlave` can now safely be stopped. Next, a watchdog timer is started to ensure that a component goes down at the latest at the timeout time, even if some of the managed tasks refuse to cooperatively stop. Three steps are necessary to cooperatively stop all managed tasks. First, the call `blocking(false)` releases all blocking waits (caused by pending requests on communication ports) inside of managed tasks. Thus, all managed tasks are able to leave their current loop and to stop the corresponding thread. Second, the call `cancel_task(baseTask)` signals all managed tasks to leave their internal loop and to stop the corresponding thread. Finally, the method `wait_task(baseTask)` blocks the calling thread till all managed tasks have stopped their internal threads. If meanwhile a timeout occurs, the callback `timerExpired` from `ShutdownTimer` is called. This callback method releases all entries (bound by the current component) from the naming service, closes the `NamingHelper`, stops the component's internal reactor and finally exits the execution context of the component. All tasks inside of the component which are not finished yet are simply killed with a SIGTERM signal. On the other hand, if all tasks stop within the timeout time, the method `wait_task(baseTask)` is returned and the timer is cancelled by the call `stopTimer`. In this case all component's internal resources are already down and thus the component's infrastructure can be safely cleaned up. Thereby, the server-initiated-disconnect handler is closed and the reactor is stopped. All internal monitors must be unblocked to prevent blocking waits inside of the destructors from service providers. Finally, the local `NamingHelper` instance is closed and the `StateSlave` is deleted. At this point all internal resources are cleaned up and the corresponding memory is freed. Thus, the execution context of the component can be safely left without the risk of memory leaks.

Chapter 4

Application Example for the State Automaton Usage

The usage of the new features in state pattern are demonstrated on a simple source code example in listing 4.1.

```
/*
 * state-pattern-example.cpp
 *
 *  Created on: 21.04.2011
 *      Author: alexej
 */

#include <smartSoft.hh>

class MyStateChangeHandler : public CHS::StateChangeHandler
{
public:
  void handleEnterState(const std::string &SubState) throw () {
    std::cout << "enter substate " << SubState << std::endl;
  }
  void handleQuitState(const std::string &SubState) throw () {
    std::cout << "quit substate " << SubState << std::endl;
  }
};

class MyInitializationTask : public CHS::ManagedTask
{
private:
  CHS::StateSlave *state;
public:
  MyInitializationTask(CHS::StateSlave *state)
  : state(state) {   }

```

```
29    int on_execute() {
30      state->acquire("init");
31      // TODO: perform individual initialization here...
32      state->release("init");
33      // break up the loop by returning != 0
34      return 1;
35    }
36  };
37
38  int main(int argc, char *argv[])
39  {
40    try {
41      // initialize component's internal infrastructure
42      CHS::SmartComponent comp("StateDemoComponent", argc, argv);
43
44      // initialize state-change handler and state-slave
45      MyStateChangeHandler state_handler;
46      CHS::StateSlave state(&comp, state_handler);
47
48      // configure and activate the state-slave
49      state.defineStates("Active", "active");
50      state.setUpInitialState("Active");
51      state.activate();
52
53      // component specific initialization comes here
54      MyInitializationTask init(&state);
55      init.start();
56
57      // switch generic state automaton into the Alive mainstate
58      state.setWaitState("Alive");
59      // start event handling (of the internal Reactor)
60      comp.run();
61    } catch (std::exception &ex) {
62      std::cout << ex.what() << std::endl;
63    } catch (...) {
64      std::cerr << "Uncaught exception..." << std::endl;
65    }
66
67    return 0;
68  }
```

Listing 4.1: Name_Request_Reply.h

The example in listing 4.1 consists of the following parts. The class `MyStateChangeHandler` (line 10) implements a simple version of a state-change-handler with the two callback methods, which simply prints out the currently activated and deactivated substates on standard output. The class `MyInitializationTask` (line 21) defines a task which is responsible to initialize component's internal resources. This tasks uses the state pattern to lock the substate `Init` as long as the initialization procedure lasts. Finally, the `main` method demonstrates the usage of a component in SMARTSOFT including the initialization of a `StateSlave` port.

The internal details of the initialization procedure (lines 42-60) are illustrated in the figure 3.4 and are described in section 3.4. By creating an instance of a `SmartComponent` class (line 42) the internal infrastructure of a component (i.e. including the connection to a naming-service) is initialized. In lines 45-46 an instance of the state-change-handler implementation is created and passed as reference, together with a reference to the `SmartComponent` instance, to the constructor of the `StateSlave` class. Thus, the `StateSlave` becomes a service of the component and calls the callback methods of `MyStateChangeHandler`.

Next, the `StateSlave` is parametrised individually for the example component (lines 49-51). In this case, a single user-defined mainstate `Active` with the substate `active` is defined. Thus, the component's individual state automaton consists of the mainstates `Active` (including the substates `active` and `nonneutral`) and `Neutral` (including the substate `neutral`). Optionally, the method `setUpInitialState` can be used to predefine the mainstate which is automatically activated after the `StateSlave` has successfully performed the `Alive` command (line 58). The internal handling of state-change-requests in the `StateSlave` is activated by the `activate` method (line 51).

The lines 52-56 are a suitable place to initialize all internal resources of a component. These resources are, for example, hardware drivers, software libraries, user-defined tasks, communication ports, etc. The initialization of these resources is best executed in one or several separate task(s) as demonstrated in the class `MyInitializationTask` (lines 21-36). The advantage of this separation is that the main method is kept very generic and can be completely generated from a component model (i.e. in the MDSD Toolchain [3, section 5]).

In line 55 the initialization task is started. This call returns immediately after the internal thread of the task is started. The initialization task locks the substate `init` and holds it as long as necessary to initialize all component's resources. The call `state.setWaitState("Alive")` (line 58) blocks the calling thread (in this case the main thread) till the substate `init` is released. Finally, if the state changes into the mainstate `Alive` (resp. switches further into the mainstate `Active`) this method unblocks and the `run` method (line 60) starts the internal handling of events in the component.

The demonstrated structure is not static, but can be modified according to component specific requirements. For example, the call `state.setWaitState("Alive")` (line 58) can be placed inside of one of the component's internal coordination threads or any other reasonable place inside of the component. A catch of the exception in line 62 can be interpreted as fatal error and thus the `StateSlave` can be commanded to switch into the mainstate `FatalError`. This can be reached by calling `state.setWaitState("FatalError")`.

Appendix A

Concept slides for a generic state automaton in a component

The concept for a generic state automaton is originally described in full length in the set of slides (attached in the following). This set of slides is structured in three parts. In the first part, the state automaton of RT-Middleware is presented as a case survey. This state automaton implements the RTC[1] standard by OMG. The state automaton is adjusted to the needs from typical use-cases in SmartSoft. As a result a concept for a generic state automaton is presented in the second part. The generic state automaton can be used on the component and the task level. In the third part, the integration of the component's generic state automaton into the state pattern of SmartSoft is presented. Additionally, a concept for the integration of an external state chart into the generic state automaton is shown.

[1] `http://www.omg.org/spec/RTC/1.0/`

A Generic State Automaton for a Service Robotic Component

M.Sc. Alex Lotz, M.Sc. Andreas Steck and Prof. Dr. Christian Schlegel

February 2010

Technik
Informatik & Medien
Hochschule Ulm
University of
Applied Sciences

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT

Overview

1. Introduction
   - Survey
   - RT-Middleware and OpenRTM
2. Concept of a generic state automaton in SMARTSOFT
   - Internal structure of a component in SMARTSOFT
   - SmartTasks in SMARTSOFT
   - Generic state automaton
3. Parameter- and Status-Port in SMARTSOFT
   - Parameter Port
   - Status Port

Lotz, Steck, Schlegel A Generic State Automaton for a Component

Outline

Introduction

A concept check questions 1

- Which internal structures are needed inside a component?
- Which structure and which states are needed by the state automaton to coordinate a component's lifecycle and activities?
- How do we describe states? Do we need different representations for (technical) functionality and business logic (e.g. scripting-languages for business logic)?
- How is the interaction between communication patterns and the state automaton in a component? Do we need to consider blocking method calls and do we need to release them when requesting state changes?

Introduction

A concept check questions 2

- What is the appropriate level of insight into the component-internal state automaton from the outside (e.g. what level of detail about the state of a component is reasonable)?
- Which kind of functionality is required to properly start and stop components and even the overall system?
- Which generic interfaces/ports (e.g. configuration, status queries, etc.) need to be provided at a component?

Outline

Case Survey: Robot Technology (RT) - OMG

Overview

OpenRTM-aist[1] OMG[2]

Links to RT-Middleware and - RTC specification

OMG - RT Component Specification: `http://www.omg.org/spec/RTC/1.0/`

RT-Middleware (OpenRTM-aist): `http://www.openrtm.org/OpenRTM-aist/html-en/`

National Institute of AIST: `http://www.aist.go.jp/`

[1] `http://www.openrtm.org/`
[2] `http://www.omg.org/`

Introduction
Concept of a generic state automaton in SmartSoft
Parameter- and Status-Port in SmartSoft
Survey
RT-Middleware and OpenRTM

RT-Middleware

Terminology

Basic Concepts:

Component Profile: meta information of a component (component description, communication ports, etc.)

Activity: denotes a state automaton that executes an algorithm in its `onExecute` method

Execution Context: abstract expression of a thread – strict separation between execution of an algorithm and the corresponding business logic

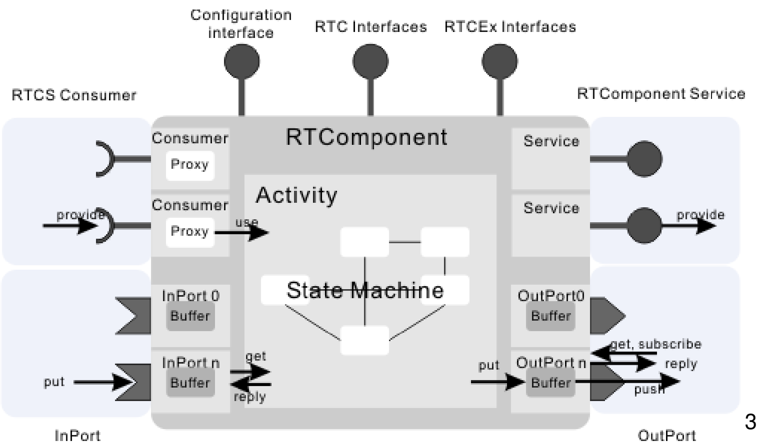
RT-Component: a module which implements the business logic, specifies the lifecycle and define the used ports and services

Lotz, Steck, Schlegel
A Generic State Automaton for a Component

Introduction
Concept of a generic state automaton in SmartSoft
Parameter- and Status-Port in SmartSoft

Survey
RT-Middleware and OpenRTM

RT-Middleware
Parts of a component 1



3

3 Figure reference:
http://www.openrtm.org/OpenRTM-aist/html-en/Documents2FRT-Middleware20Overview.html

Lotz, Steck, Schlegel — A Generic State Automaton for a Component

Introduction
Concept of a generic state automaton in SmartSoft
Parameter- and Status-Port in SmartSoft

Survey
RT-Middleware and OpenRTM

RT-Middleware
Parts of a component 2

Each component posses a uniform interface:

- Configuration Interface: to modify parameters from outside the component at runtime
- RTC Interface: to trigger and poll the state automaton
- RTCEx Interface: to manage the execution context

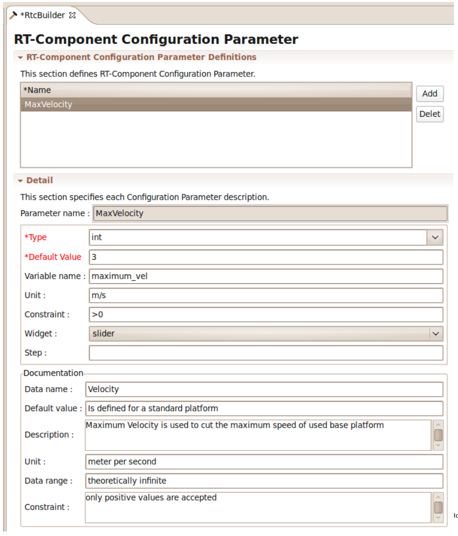
User-defined communication can b e implemented via:

- Data Ports: Communication for simple data types (Data-Flow Port)
- Service Ports: RPC interface based on CORBA

Lotz, Steck, Schlegel — A Generic State Automaton for a Component

RT-Middleware
Configuration Interface 1

- Screenshot illustrates initialization of parameters and according tags
- Attached documentation fields give further hints on the parameter's semantics
- These parameters can be modified at runtime via the configuration interface
- Problem: Only basic data types are allowed



Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Survey
RT-Middleware and OpenRTM

RT-Middleware
Configuration Interface 2

- Basis data types are not sufficient as parameters (see examples for parameters on slide 80)
- Mapping of composite data structures onto several commands with basic data types is not an atomic operation. It is difficult to command consistent configuration sets.
- In addition to a configuration interface a generic command interface is needed, which must be manually implemented by using service ports.

Lotz, Steck, Schlegel
A Generic State Automaton for a Component

RT-Middleware

Configuration Interface 3

- In some cases parameters of the same type must be collected in a FIFO queue to consecutively execute them in a component (e.g. several goal regions in a planner component), which is not possible in OpenRTM.
- Advantage of simplified approach: the direct mapping of parameters on internal variables in a component allows to simply request for changes of these variables through the same configuration interface

RT-Middleware

Basic idea behind Execution Context and RT-Component 1

- The *Execution Context* supports the separation of the business logic and worker thread[4].
- This allows to execute these business logic blocks in different arrangements, either sequentially or concurrently depending on assignment and parametrisation of execution contexts.
- Realtime processing is achieved by sequential execution of several business logic blocks [ASKK05] (cyclic executive).
    - Problem: Realtime scheduling strategies are being ignored (like rate monotonic scheduling, or earliest deadline first)

[4]Reference: `http://www.omg.org/spec/RTC/1.0/` on pages 22-23

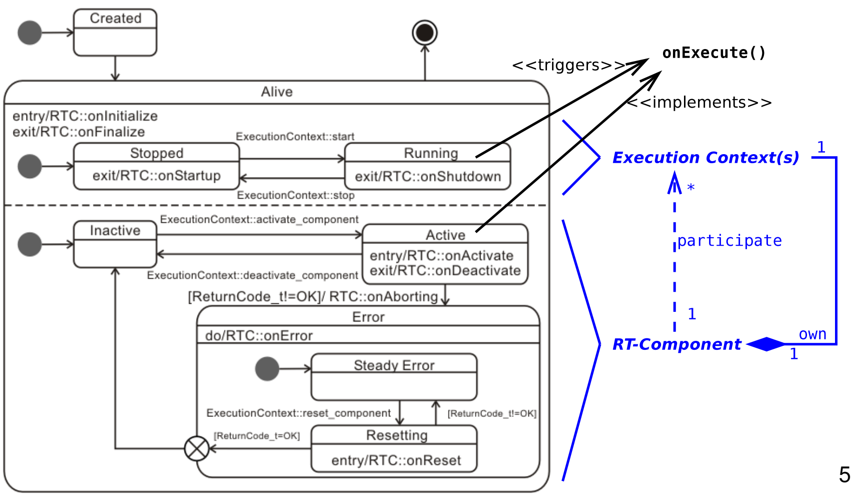
RT-Middleware

Basic idea behind Execution Context and RT-Component 2

- The synchronization of several components is only possible by sequential execution in a single execution context.
  - By contrast, in SMARTSOFT the synchronization is realized by communication patterns (compare `getUpdateWait()` for example).
- In general, an RT-Component participates with different execution contexts, implements the `onExecute` method an specify its lifecycle.
- At a certain point in time an execution context belongs to one particular component, but is able to call several `onExecute` methods from corresponding component-participants.
- An execution context can be configured either as periodic (with a certain period) or sporadic (full load) and additionally can be started and stopped from the outside.

RT-Middleware

State Automaton in a RT-Component 1



[5]

[5] Figure reference:
www.openrtm.org/OpenRTM-aist/html-en/Documents2FProgramming20RT-Component.html

RT-Middleware
State Automaton in a RT-Component 2

- The state automaton in RT-Middleware combines the RT-Component with one particular execution context inside of the state `Alive`.
- An RT-Component is at first passive and becomes active by participating in an execution context.
- The state automaton in the lower region of the `Alive` state is implemented in a RT-Component.
- The state automaton in the upper region of the `Alive` state is either linked to an owned or external execution context.

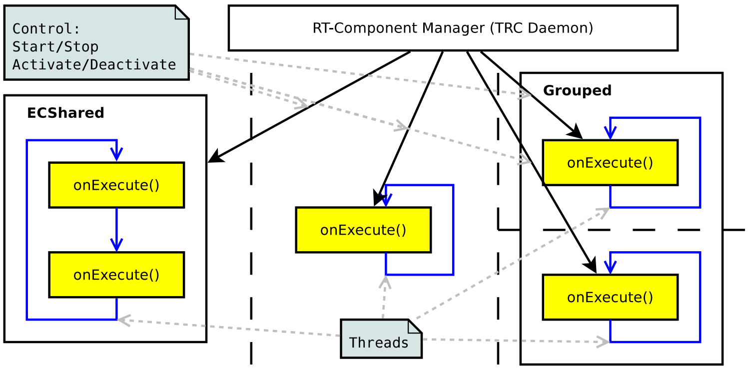
Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Survey
RT-Middleware and OpenRTM

RT-Middleware
State Automaton in a RT-Component 3

- Each RT-Component implements business logic inside of its `onExecute` method.
- The method `onExecute` is triggered only if the RT-Component is in its Alive state and a corresponding execution context is in its Running state.
- Both states can be commanded at runtime form the outside of a component through the generic RTC interface of the component.
- At runtime the method `onExecute` is triggered as a callback function from a corresponding execution context.

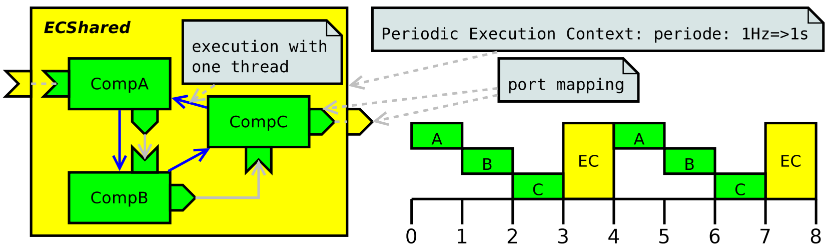
Lotz, Steck, Schlegel A Generic State Automaton for a Component

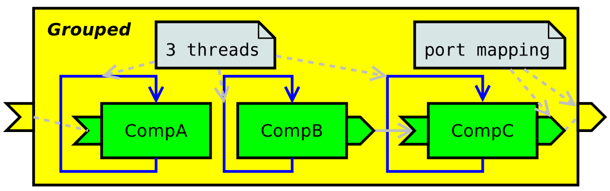
RT-Middleware
Composite Components 1



On the left is a shared execution context between two components, in the middle a simple component with one execution context and on the right a group of components each using a separate execution context.

RT-Middleware
Composite Components 2





combination of several components in a group

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT

Survey
RT-Middleware and OpenRTM

RT-Middleware
Composite Components 3

- Each RT-Component defines an own state automaton with one `onExecute` method.
- An execution context executes either one particular component or is shared by several components (ECShared). Thereby, the compoonents are executed in a sequence.
- Grouped components are not allowed to share memory, but must communicate through data and service ports.
- The group of components is not allowed to define own communication ports, but must publish already available ports from components in its internal group.

Hochschule Ulm

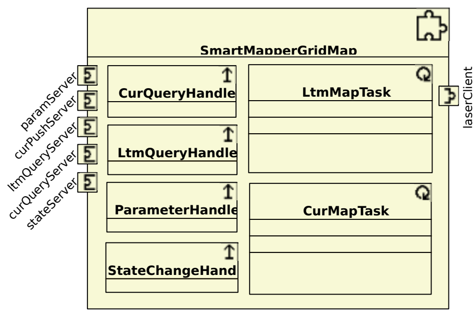
Lotz, Steck, Schlegel   A Generic State Automaton for a Component

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT

Survey
RT-Middleware and OpenRTM

RT-Middleware
Mapper Example from SMARTSOFT

- Two independent tasks calculate the longterm, resp. the current map
- The two tasks must share internal resources (like shared set of parameters, internal data structures for the current map which is used to calculate the longterm map, etc.)
- The two tasks must be activated and deactivated independently.



In the following some possible solutions are presented and evaluated which result from eMail contact with a RT-Middleware maintainer.

Hochschule Ulm

Lotz, Steck, Schlegel   A Generic State Automaton for a Component

RT-Middleware

Mapper Example - First Suggestion

"Implement your component so that it calculates the long-term map first, outputs it, then calculates the current map based on the long-term map, and outputs that. Do all this in sequence in onExecute()." (suggested by RT-Middleware maintainer, personal eMail communication)

Resulting problems:

- Execution of the LTMap and CurrentMap are not independent of each other.
- Thus, a separate execution frequency is not possible.
- It is not possible to activate and deactivate LTMap and CurrentMap independent of each other.

Introduction
Concept of a generic state automaton in SmartSoft
Parameter- and Status-Port in SmartSoft
Survey
RT-Middleware and OpenRTM

RT-Middleware

Mapper Example - Second Suggestion

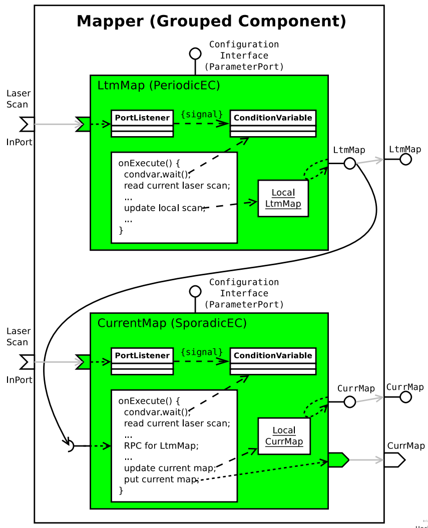
- *"Create two separate components, one creating the long-term map and one creating the current map. Give the long-term map component a service port to access the information necessary for the current map, and make the current map component use it. Put them into a composite component with only the map outputs exposed."* (suggested by RT-Middleware maintainer, personal eMail communication)
- *"Same as above, but use a pull-based data port to pull the latest long-term map information from the current map component."* (suggested by RT-Middleware maintainer, personal eMail communication)

Lotz, Steck, Schlegel
A Generic State Automaton for a Component

RT-Middleware
Mapper Example - Second Suggestion

design and model problems

- Duplicated port for Laser Scans
- Internal communication inefficient over network
- Parametrization spread over two components (synchronization problem)
- No separation between internal and external views (public/private)
- Synchronized reception of data must be error prone programmed using condition variables



RT-Middleware
Mapper Example - Third Suggestion

"Use your own threads internally within a single component, one for the long-term map and one for the short-term map. onExecute() would talk to these threads to get the latest data to output each time it gets called." (suggested by RT-Middleware maintainer, personal eMail communication)

Resulting problems:

- The framework is circumvented
- Thread is hidden from the outside of a component
- The concept of execution contexts is ignored
- Thread is not controlled by the state automaton (or the control must be error prone implemented)

RT-Middleware

Additional recurring issues

- Each execution context can be directly controlled from the outside. It is not possible to hide execution contexts to reduce the overall complexity. Thus, a user needs deep knowledge about internal system structures to know when which execution contexts are allowed to be activated or not.
- Active handlers for consecutively execution of requests (like SmartProcessing patterns in SMARTSOFT) are missing and must be error prone, manually implemented.

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Survey
RT-Middleware and OpenRTM

RT-Middleware

Additional recurring issues

- Event mechanisms (like Event Pattern in SMARTSOFT) must be emulated (costly, time-consuming, error prone)
- Interactions between the state automaton and data/servcie ports are not considered and must be emulated.

Lotz, Steck, Schlegel
A Generic State Automaton for a Component

Lesions
Conceptual Problems 1

- The separation of business logic and execution thread leads to fine grained components. This resembles active classes and does not reduce the overall complexity in a system.
- The resulting components are tightly specialized on the individual usage in a certain system and are difficult to reuse in other (different) systems.
- Separation of internal and external views is not possible.
- Communication mechanisms are too generic. Certain communication pattern semantic must be emulated on top of these mechanisms.

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Survey
RT-Middleware and OpenRTM

Lesions
Conceptual Problems 2

- Several use cases in robotics require the execution of several tasks in a component, which is not directly possible in RT-Middleware and must be implemented as:
  - Composite components (no internal shared memory possible)
  - Several execution contexts in a single component. Inside of `onExecute` an `ec_id` must be evaluated to decide which execution context is currently active, for example:

```
if(ec_id == 1) do A;
else if(ec_id == 2) do B;
```

Lotz, Steck, Schlegel A Generic State Automaton for a Component

Lesions
Problems in the Toolchain and the implementation

Eclipse based Toolchain

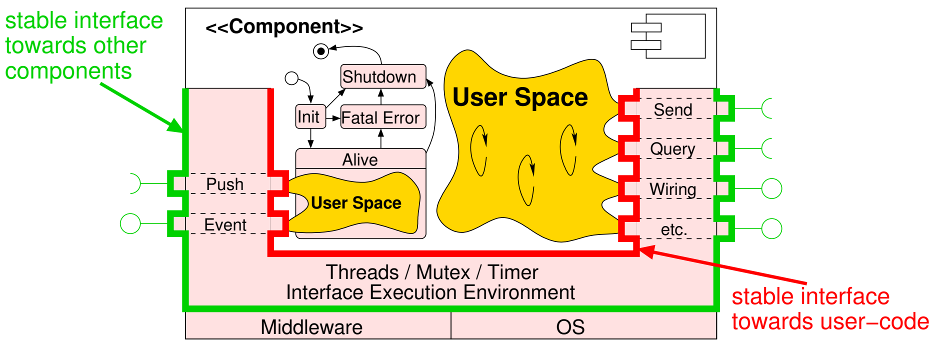
- no proper model used for code generation (just a GUI formular based approach)
- no proper code generation but a simple comment in and out of in a static code structure
- after a further generation run, the sources must be merged by a merge-tool

OpenRTM implementation

- No abstraction level to hide CORBA details in service ports
- semantic of communication ports way too simple

Outline

Parts of a Component in SMARTSOFT



Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Internal structure of a component in SMARTSOFT
SmartTasks in SMARTSOFT
Generic state automaton

Parts of a Component in SMARTSOFT

State Automaton: Provides generic base states of a component (*Init*, *Alive*, *Shutdown* and *FatalError*). These states are traversed during execution of a component. Inside of *Alive* state the user can define own component specific states.

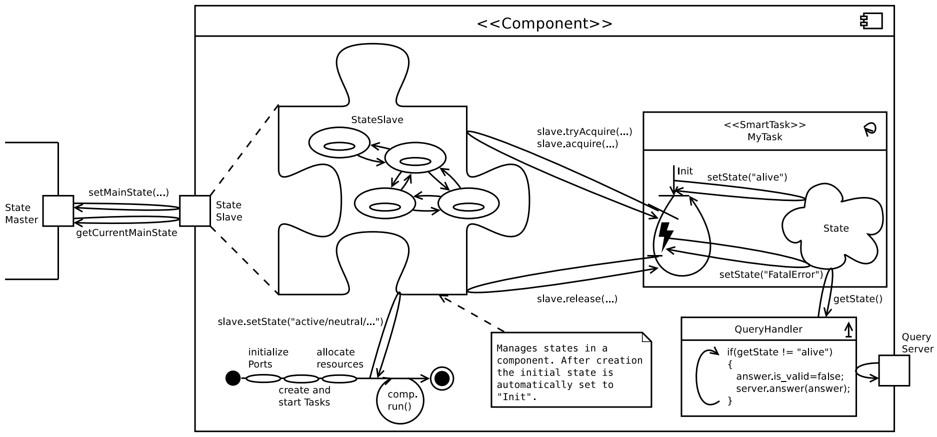
Service Ports: Implements the communication patterns of SMARTSOFT. They are used for communication between components as well as for orchestration of components from the sequencing layer

Tasks: Allows to implement concurrent activities in a component.

OS Abstraction: Mutex, Condition Variable, Semaphore, etc.

User-Code: Code part implemented by component developer (can contain arbitrary libraries)

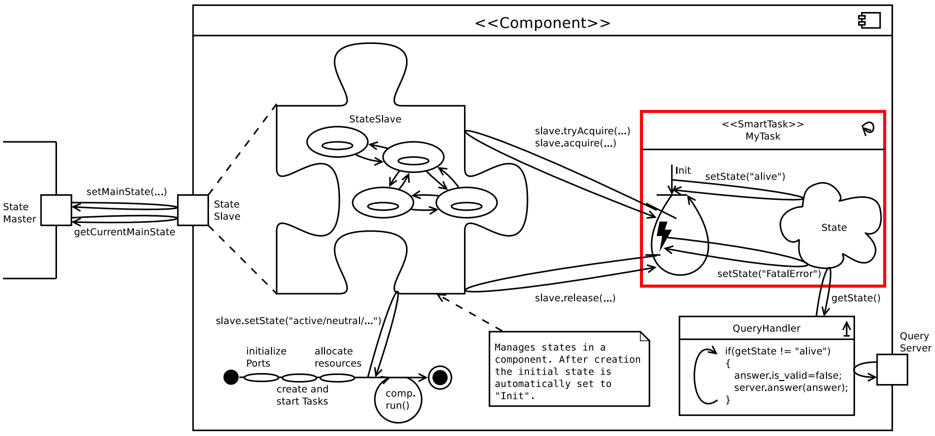
Overview for the State Automaton Concept



Outline

1. Introduction
   - Survey
   - RT-Middleware and OpenRTM
2. Concept of a generic state automaton in SMARTSOFT
   - Internal structure of a component in SMARTSOFT
   - SmartTasks in SMARTSOFT
   - Generic state automaton
3. Parameter- and Status-Port in SMARTSOFT
   - Parameter Port
   - Status Port

Concept overview for states in a SmartTask



SmartTasks in SmartSoft

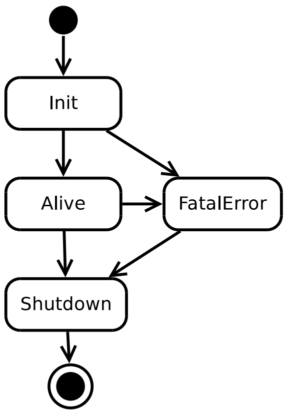
Overview

- SmartTasks allow to realize concurrent execution inside of components.
- A SmartTask traverses the states *Init*, *Alive* and *Shutdown* during its lifetime. Occurrences of critical (locally not solvable) errors can be outlined by the state *FatalError*.
- The current state of a SmartTask can be accessed and evaluated in a component.
- The handlers of communication patterns can use the state in their execution.
- Explicated states are the basis for *Monitoring* and *Introspection*.

States in a SmartTask

Semantics of the states in a SmartTask 1

1. ***Init:*** A task is prepaired for its execution. Required resources are initialized here.
2. ***Alive:*** A tasks is fully initialized and is ready to execute.
3. ***Shutdown:*** Clean up procedures that are executed on shutdown of a task.
4. ***FatalError:*** This states indicates critical errors in a SmartTask that are not solvable by regular error handling strategies locally inside of the task. The task can be only stopped or restarted from here.



States in a SmartTask

Semantics of the states in a SmartTask 2

- The states of a task are realized inside of the class SmartTask. Member functions provide access to the states (to modify and read them).
- All four states can be read from the outside the task (within a component) as well as from the inside the task by using public member methods of the class SmartTask.
- During the creation of a SmartTask the state *Init* is set as the first state per default. The states *Alive*, *Shutdown* and *FatalError* are only allowed to be modified from within the task itself by using protected member functions of the class *SmartTask*.
- A component developer is able to define the points, where the state changes are triggered, according to his needs. This enables a component developer, for example, to exactly define when the internal resources of a task are fully initialized and the task is thus ready to execute (by switching into the state *Alive*).

States in a SmartTask

Relation between a task and its transitions 1

- A SmartTask provides the methods `start()` and `stop()` to respectively start and stop the internal activity of the task (even multiple times as long as the task is in its *Alive* state).
- The execution of a task can be started first after its initialization is completed.
- Therefore the method `start()`, for example, can check whether the task is in its *Alive* state and only in this case start the internal activity.
- The state change into the state *Alive* can be triggered either from the constructor of the derived class or from an other arbitrary point in the implementation of the derived class.

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Internal structure of a component in SMARTSOFT
SmartTasks in SMARTSOFT
Generic state automaton

States in a SmartTask

Relation between a task and its transitions 2

- The activity of a task is stopped by calling the `stop()` method. The task remains to be in the *Alive* state.
- If however the resources of a task are cleaned up, the task must switch into the state *Shutdown*. This ensures that the task is not started again without the initialized resources of this task.
- If the task is switched into the *FatalError* state, its resources remain allocated. The activity of the task is stopped and the task is not able to start again. The only escape is to switch into *Shutdown* state and to clean up task's resources.

Lotz, Steck, Schlegel A Generic State Automaton for a Component

States in a SmartTask

Overview for all possible transitions between states of a SmartTask

The transitions between states are performed according to the following table. Forbidden transitions are prevented by returning a corresponding return value in the member functions of a SmartTask.

	target state			
current state	Init	Alive	Shutdown	FatalError
Init	-	X	-	X
Alive	-	-	X	X
Shutdown	-	-	-	-
FatalError	-	-	X	-

Keys:

- ***[X]:*** Transition is allowed
- ***[-]:*** Transition is not allowed

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Internal structure of a component in SMARTSOFT
SmartTasks in SMARTSOFT
Generic state automaton

States in a SmartTask

Interactions between the states in a SmartTask and Communication Patterns in SMARTSOFT

Handlers of Communication Patterns:

- In the implementation of the handlers the current state of a SmartTask can be evaluated to implement corresponding handler behavior.
- *Example:* In a query handler the current state of a task can be evaluated such, that the handler returns a valid answer only in the case when the state equals *Alive*. Otherwise a valid flag (e.g. `is_valid=false`) in the answer communication object is set to false.

Lotz, Steck, Schlegel
A Generic State Automaton for a Component

States in a SmartTask

Difference between a regular error and a fatal error in a SmartTask

Regular Error: Is a deviation from a regular execution in a component, which can be resolved locally and is not critical for the overall execution of the task.

- Example: If the path planning component does not find a path due to occupied target region, a corresponding (regular) flag in a communication object is set. This enables a CDL component to recognise this situation and react in a reasonable way. The planner task can continue to execute (without stopping its internal activity) and thus remains in its *Alive* state.

Fatal Error: Is a critical error at runtime, which can not be solved in the task and prevents the task from continuing its execution. In this case the task is switched into the *FatalError* state. If the task is additionally critical for the overall execution of its component, the component as well is switched into the *FatalError* state (as presented later).

- Example: A fatal erro in a GUI task is not critical for a component and affects only the task.

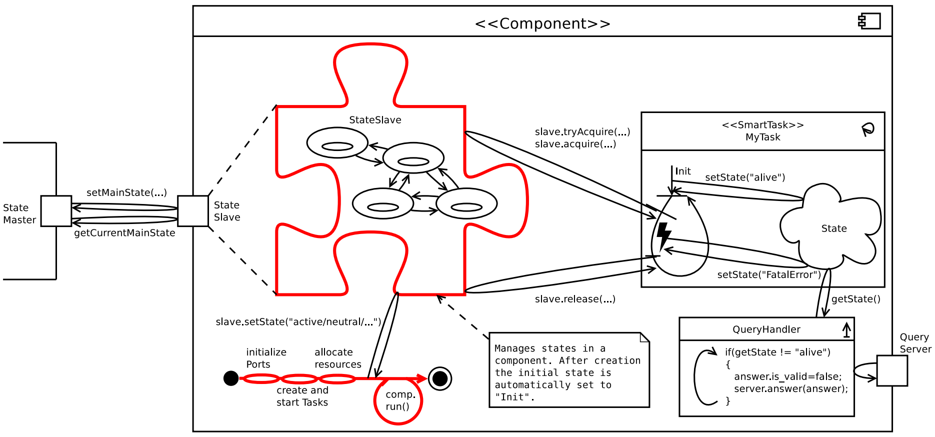
Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Internal structure of a component in SMARTSOFT
SmartTasks in SMARTSOFT
Generic state automaton

Outline

Lotz, Steck, Schlegel
A Generic State Automaton for a Component

Generic State Automaton in a Component

Concept overview



Generic State Automaton in a Component

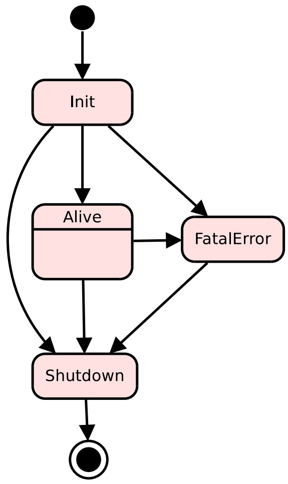
Concept overview

- A generic state automaton in a component explicates the lyfecycle of this component by the four base states *Init*, *Alive*, *Shutdown* and *FatalError*.
- This state automaton is included into the State Pattern of SMARTSOFT.
- Inside of the state *Alive* further user-defined states can be defined. Even an integration of an external state-chart (e.g. from IAR VisualState) is possible.
- In general, a state automaton allows to activate / deactivate different behaviors of a component, by indirectly activating and deactivating corresponding tasks.

Generic State Automaton in a Component

Semantics of the four base states

- ***Init:*** A component is prepared for its execution. Thereby the infrastructure and the resources of the component are initialized.
- ***Alive:*** Is a pseudo state (and a placeholder for a user-defined state automaton) that indicates a running component.
- ***Shutdown:*** A component is in process of shutdown, where all resources are freed.
- ***FatalError:*** Is a critical error, which is not solvable locally in the component. This state can be resolved only by restarting the component.

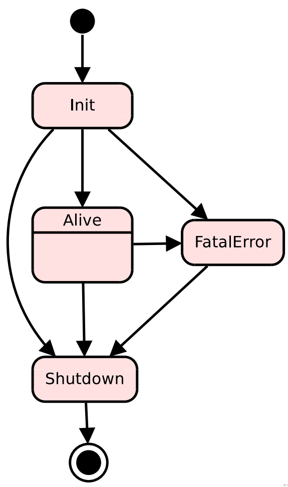


Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Internal structure of a component in SMARTSOFT
SmartTasks in SMARTSOFT
Generic state automaton

Generic State Automaton in a Component

Semantics for the transitions between the four base states 1

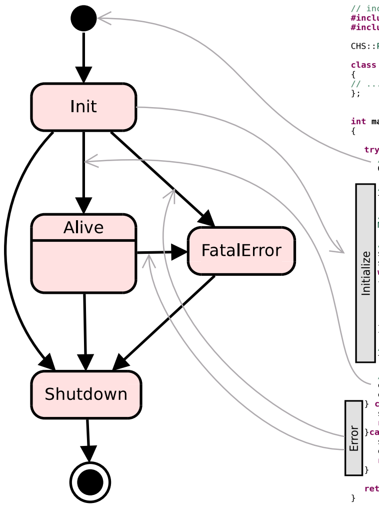
- ***Init -> Alive:*** This transition is triggered from within the component by a local method call. This enables a component developer to exactly define the point in time when the initialization of a component is completed.
- ***Init -> FatalError:*** If the initialization fails due to an unsolvable error, which prevent the component to continue its initialisation procedure, the component can be switched into the *FatalError* state by a local method call.
- ***Init -> Shutdown:*** The shutdown of a component can be triggered from the outside of a component as well as from within a component.



Generic State Automaton in a Component

Semantics for the transitions between the four base states 1

Example presented on a source code level:

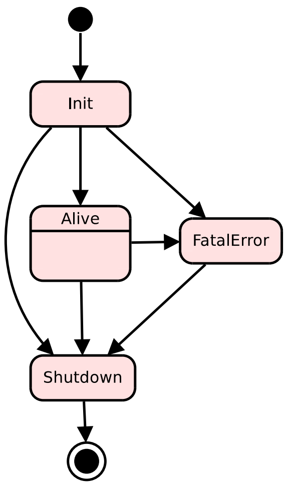



```
// include the SmartSoft framework and Communication-Object(s)
#include <smartSoft.hh>
#include <commExampleTime.hh>

CHS::PushNewestClient<CHS::CommExampleTime> *time_client;

class MyTask: public CHS::SmartTask
{
// ...
};


int main(int argc, char *argv[])
{

   try {
     // create and initialize SmartSoft-Component-Hull
     CHS::SmartComponent comp("ExampleComponent", argc, argv);

     // create and initialize component's CommunicationPort(s)
     time_client = new CHS::PushNewestClient<CHS::CommExampleTime>(&comp);

     // create Task(s) for user defined executions context(s)
     MyTask task;

     // try to (re)connect all ServiceRequestor(client) - ports (static configuration)
     std::string server = "SomeServer";
     std::string service = "SomeService";
     while(time_client->connect(server, service) != CHS::SMART_OK)
     {
        std::cout << "Connection to server: " << server << "; service: " <<
             service << "; FAILED! sleep(1) and trying reconnect..." <<
             std::endl;
        ACE_OS::sleep(1);
     }

     // start task(s) to execute
     task.open();

     // start component management (with Reactor-Event-Loop)
     comp.setState("Alive");
     comp.run();
   } catch (std::exception &e) {
     std::cerr << e.what() << std::endl;
     return 1;
   }catch(...) {
     std::cerr << "Uncought exception..." << std::endl;
     comp.setState("FatalError");
     return -1;
   }

   return 0;
}
```


Generic State Automaton in a Component

Semantics for the transitions between the four base states 2

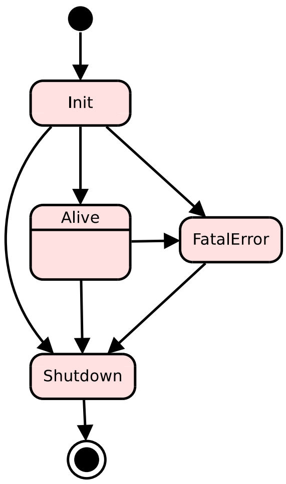
- ***Alive -> FatalError:*** A critical error at runtime, that prevents a component to further provide its service, the component can be set into the *FatalError* state by a local method call.
- ***Alive -> Shutdown:*** A component can be commanded to shutdown by switching into the state *Shutdown*. This can be triggered either from the outside or from within a component (e.g. by catching the SIGINT signal).



Generic State Automaton in a Component

Semantics for the transitions between the four base states 3

- ***FatalError -> Shutdown:*** A component being in the *FatalError* state is not able to resolve this problem locally. Thus, the component requires help from a higher level (outside of this component). The only way to escape this state is to switch into the state *Shutdown* (e.g. triggered by a scenario coordination component on the sequencing layer). Again, an error which can be resolved by a regular error handling strategy locally in the component must be solved locally and does not result in a *FatalError* state. A fatal error is, for example, if the communication basis of a component crashes down.



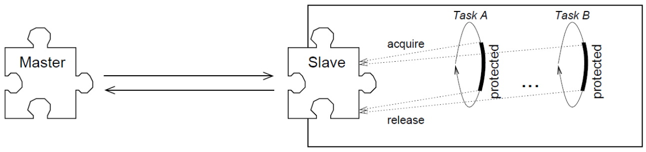
The State Pattern in SMARTSOFT

Overview

- The State Pattern [Sch04] in SMARTSOFT implements a master-slave relationship.
- The State Pattern provides a generic interface on a component to set the component into different modes (the mainstates) from the outside in a generic way.
- By evaluating the internal states (the substates) inside a component, the internal tasks of the component can be activated and deactivated.
- A mainstate is a mask for a valid combination of substates. Both state types are defined during design time of a component.

The State Pattern in SMARTSOFT

Master-slave relationship 1

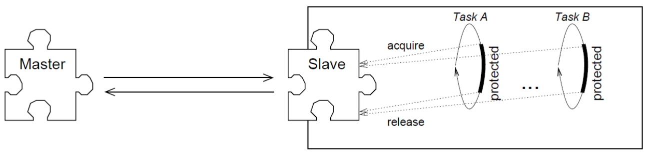


State Slave

- A slave provides an interface to the outside of a component to be able to set the component into different modes (mainstates).
- A task in a component can lock a substate by calling the `acquire()` method from the slave and respectively unlock a substate by calling its method `release()`.
- This protects tasks to be interrupted at unsuitable points of execution.

The State Pattern in SMARTSOFT

Master-slave relationship 2



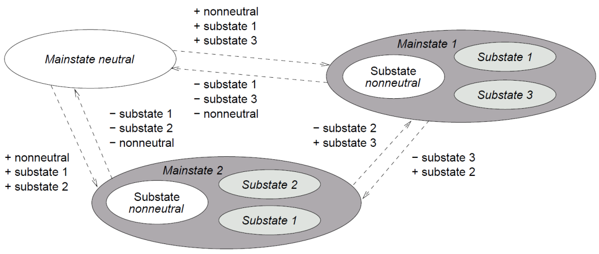
State Master

- Is used to command state changes in a state slave.
- A synchronous call from the state master ensures that the state slave successfully completes a state-change-request.
- A state-change can be performed only after all locked substates (which are affected by this state-change) are released.

The State Pattern in SMARTSOFT

The concept of mainstates and substates

- A mainstate contains several substates
- A state master commands only mainstates
- A state slave provides substates within a component



- At a certain point in time only one mainstate can be active. If a mainstate is activated, its substates are activated as well. In case substates are contained in the current and the new mainstate during a state-change, these substates remain active and are not affected by the state-change. Substates, which are not included in the new mainstate are deactivated.

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Internal structure of a component in SMARTSOFT
SmartTasks in SMARTSOFT
Generic state automaton

The State Pattern in SMARTSOFT

Properties of the State Pattern 1

- A change to any valid mainstate is allowed (independent of the previous mainstate). A mainstate is some kind of a mask for a valid combination of substates.
- A state-change can be completed only if all substates, which are not included in the next mainstate, are unlocked. Substates which are contained in both mainstates must not be unlocked.
- This ensures that all corresponding activities are in a safe mode where a state-change can be performed without negative impacts.
- During a state-change, all substates which are not included in the next mainstate, can not be locked again (after a release is called).

Lotz, Steck, Schlegel A Generic State Automaton for a Component

The State Pattern in SMARTSOFT

Properties of the State Pattern 2

- A state-change into the mainstate *Neutral* (and only *Neutral*) can be instantly enforced by the special command *deactivate*. Thereby, all blocking calls in a component, which are caused by pending requests on communication patterns (e.g. `getUpdateWait(...)`) are instantly unblocked. This enables all tasks to release the corresponding substates as fast as possible.
- For each activated substate the callback method `handleEnterState` of the `StateChangeHandler` is called. For each deactivated substate the callback method `handleQuitState` of the `StateChangeHandler` is called.
    - These callback methods allows to acquire / release resources, connect / disconnect service requestors, open / close files, etc.

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Internal structure of a component in SMARTSOFT
SmartTasks in SMARTSOFT
Generic state automaton

The State Pattern in SMARTSOFT

Properties of the State Pattern 3

- The mainstate *Neutral* containing the substate *neutral* is automatically available from the beginning in the state pattern.
- State pattern user can define individual mainstates, each with a separate combination of substates.
- Each user-defined mainstate automatically contains the substate *nonneutral*. This allows to start activities as soon as the mainstate *Neutral* is deactivated (independent of the next mainstate).

Lotz, Steck, Schlegel
A Generic State Automaton for a Component

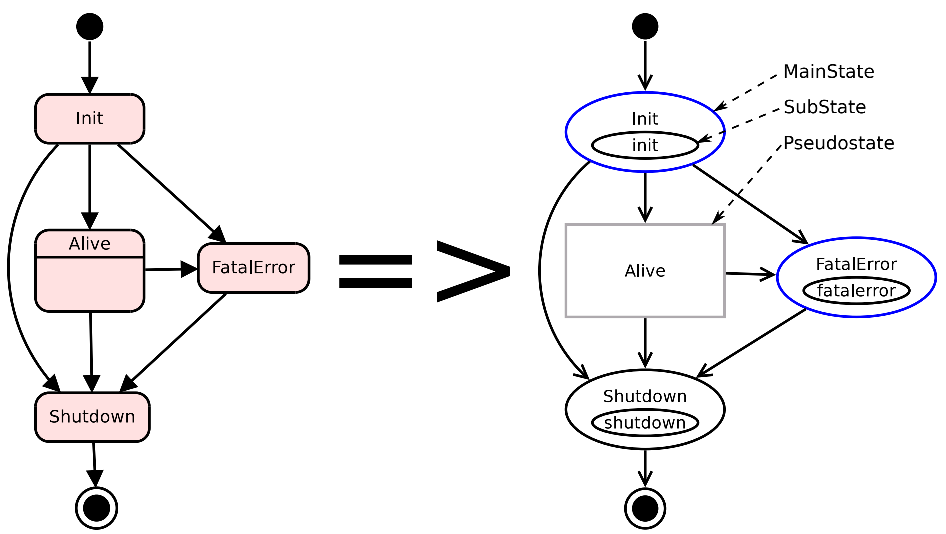
The State Pattern in SMARTSOFT

Conclusion

- In contrast to a finite state machine (where state-changes are triggered by corresponding events), the state pattern enables a developer to command any valid mainstate in a simple way.
  - Thus, the focus is not on the state-changes itself (as it is for example by StateCharts), but to activate and deactivate tasks in a component in a simple and reusable way.
- The mainstate mask hides the combination of substates and thus hides the realisation of a mainstate. A component developer can even change the internal behavior of a component (e.g. during its development) by adding/removing substates to/from a certain mainstate, without affecting the orchestration of the mainstates by a state master.
- Hence, the state master can safely command all valid mainstates without the need to know which substates are allowed to be active at certain points in time.

The Generic State Automaton in a Component

Integration into the State Pattern in SMARTSOFT



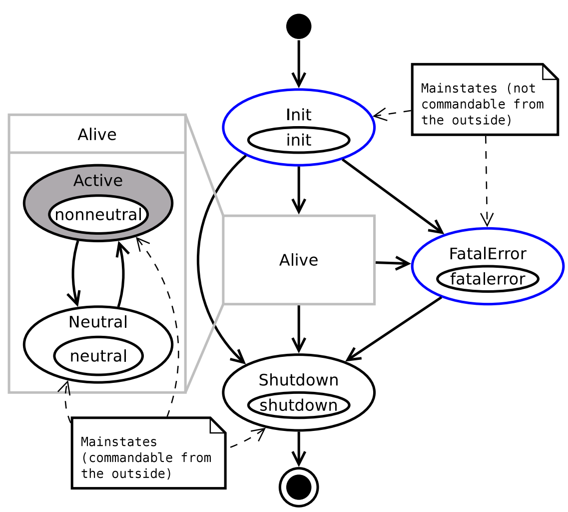
The Generic State Automaton in a Component

Integration into the State Pattern in SMARTSOFT

- The four base states of the generic state automaton are implemented as mainstates in the state pattern in SMARTSOFT. These mainstates are available from the beginning in the state slave.
- The mainstates *Init*, *FatalError* and *Shutdown* consist of exactly one substate with the same name (except the first letter, which is changed to a lowercase letter).
- The integration of the generic states into the state pattern facilitates a uniform way to request and command states from the generic and the user-defined state automatons.
- The mainstate *Alive* is a pseudo state and a placeholder for the user-defined state automaton in the state pattern.

The Generic State Automaton in a Component

Example for the combination of the generic and a simple user-defined state automatons



The Generic State Automaton in a Component

Example for the combination of the generic and a simple user-defined state automatons

- A developer has to define at least one mainstate (which automatically obtains the substate *nonneutral*).
- If the name for this mainstate is not needed to be specific, it is recommended to call it *Active*.
- A component being in the mainstate *Neutral* behaves as passive as possible (see next slide).
- A component developer specifies which mainstate is initially chosen after the mainstate *Alive* is commanded. Per default the initial mainstate is *Neutral*.

The Generic State Automaton in a Component

Guiding rules for the semantics in the mainstate *Neutral*

In the mainstate *Neutral*:

- . . . a component does not use services from other components (client ports are disconnected or unsubscribed)
- . . . a component does not require much CPU time (tasks block on corresponding substate locks)
- . . . the client ports of a component can be safely rewired (e.g. by using the Dynamic Wiring pattern)
- . . . the component can be safely reconfigured (new parameters can be set)

The Generic State Automaton in a Component

Guiding rules for the semantics in the mainstate *Neutral*

In the mainstate *Neutral*:

- ...the server ports behave as follows:
  - Query: requests are answered with an empty message (containing the flag `is_valid=false`)
  - Send: is executed as before (because no reply is available)
  - PushNewest and Event: Do not publish anything (tasks are blocked on substates)
  - PushTimed: its method `stop()` halts the service and all connected clients are informed about this with the return value "SMART_NOTACTIVATED"
  - State and Wiring: are fully functional
  - Parameter: can be used as before

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Internal structure of a component in SMARTSOFT
SmartTasks in SMARTSOFT
Generic state automaton

The Generic State Automaton in a Component

Guiding rules for the semantics in the mainstate *Init*

In the mainstate *Init*:

- As long as the component is in its *Init* state the service ports of this component are not able to provide reliable service. Thus, a remote component con only rely on the service after a component has switched into the *Alive* state.
- Even if the service ports in a component are fully initialized (after the call `component.run()`), the component can still delay the switch into the *Alive* state (e.g. to wait till the internal SICK laser device driver has fully established a serial connection)

Lotz, Steck, Schlegel
A Generic State Automaton for a Component

The Generic State Automaton in a Component

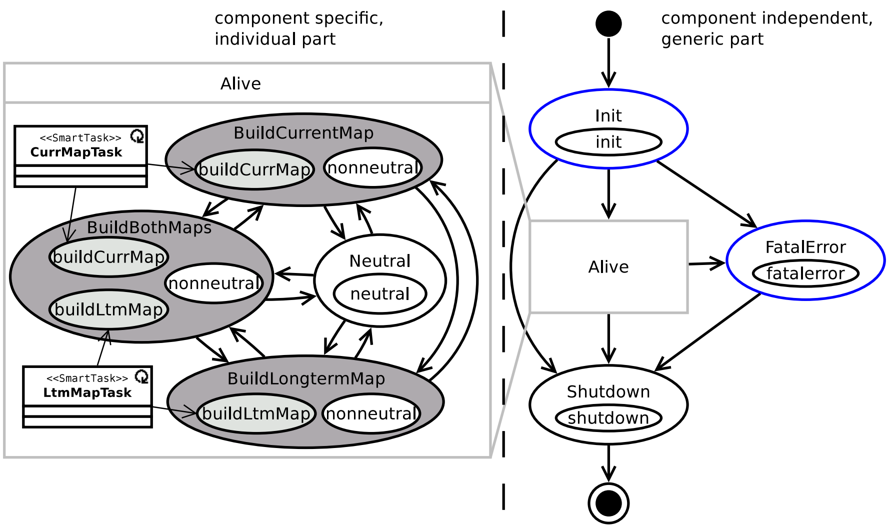
Guiding rules for the semantics in the mainstate *Shutdown*

In the mainstate *Shutdown*:

- . . . a component commands all its internal tasks to cooperatively shutdown.
- . . . all service ports of this component are deleted and cleaned up
- . . . the component's internal infrastructure is cleaned up
- . . . the component leaves its execution context

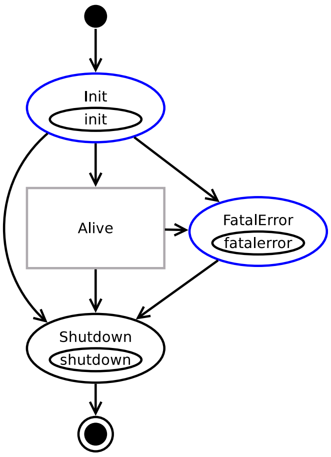
The Generic State Automaton in a Component

Example: State Automaton in the Mapper Component



The Generic State Automaton in a Component
Internal and External View

- Outside of a component all mainstates are visible, those from the generic and those from the user-defined state automaton.
- A state master is only allowed to command the mainstates *Neutral*, *Shutdown* and the user-defined mainstates.
  - The mainstate *Init* is automatically set as the very first mainstate and thus can not be set from outside a component.
  - The mainstate *FatalError* is only identifiable inside a component and can thus be only set from within a component.

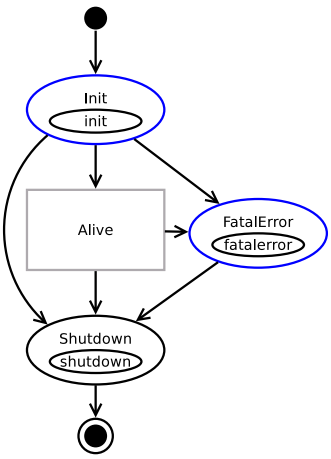


Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Internal structure of a component in SMARTSOFT
SmartTasks in SMARTSOFT
Generic state automaton

The Generic State Automaton in a Component
Internal and External View

Pseudostate *Alive*

State-changes between user-defined mainstates and the mainstate *Neutral* can not be commanded from within a component, but only from a remote state master. This defines clear responsibilities for the state master.

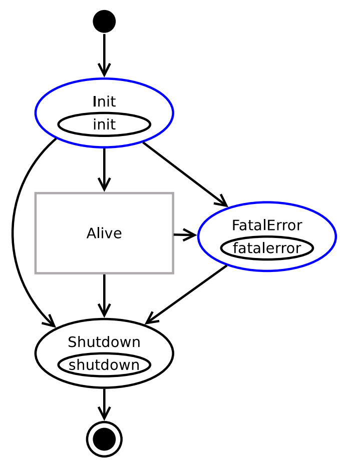


The Generic State Automaton in a Component

Internal and External View

Mainstate *FatalError* and *Shutdown*

It is not reasonable to set a *FatalError* mainstate outside a component, because if a component is identified to be faulty (e.g. by a monitoring component) it can be directly commanded to shutdown without the intermediate *FatalError* mainstate. Also, if the component is already in the *FatalError* state, the only reasonable way to resolve this is to shutdown a component from the outside by a state master.



Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Internal structure of a component in SMARTSOFT
SmartTasks in SMARTSOFT
Generic state automaton

The Generic State Automaton in a Component

Overview of all allowed transitions

	Target state					
Current state	Init	*Alive*	Neutral defined	User-	FatalError	Shutdown
Init	-/-	I/-	-/-	-/-	I/-	I/E
Alive	-/-	I/-	predef./-	predef./-	I/-	I/E
Neutral	-/-	-/-	-/E	-/E	I/-	I/E
User-defined	-/-	-/-	-/E	-/E	I/-	I/E
FatalError	-/-	-/-	-/-	-/-	I/-	I/E
Shutdown	-/-	-/-	-/-	-/-	-/-	I/E

- ***[I]:*** Transition allowed to be set (internally) within a component
- ***[predef.]*** Transition is predefined by the user during initialisation
- ***[E]:*** Transition allowed to be set externally by a state master
- ***[-]:*** Transition is not allowed

The Generic State Automaton in a Component

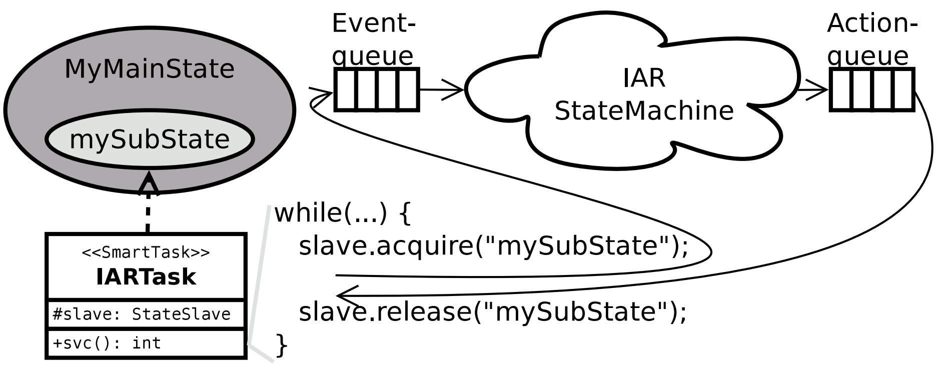
Relationship between communication patterns and the state automaton

- In the implementation of the handlers from communication patterns a substate can be checked without blocking by using the method `tryAcquire(...)` and the handler can behave according to the results.
- For example:
  - A current map is requested from a mapper component by using its query server port: If the mapper is currently deactivated (which can be checked by calling `tryAcquire("neutral")`), the handler from the query server can answer an empty map with the flag `is_valid=false`.
  - As long as the mapper component is in the mainstate *Neutral*, parameters of this component are allowed to be changed (e.g. the size for the current map can be only changed if no task currently tries to update the map).

Integration of a StateChart (1st case with tight coupling)

A StateChart inside of a SmartTasks using a substate

Example: Include an external StateMachine from IAR VisualState



Integration of a StateChart (1st case with tight coupling)

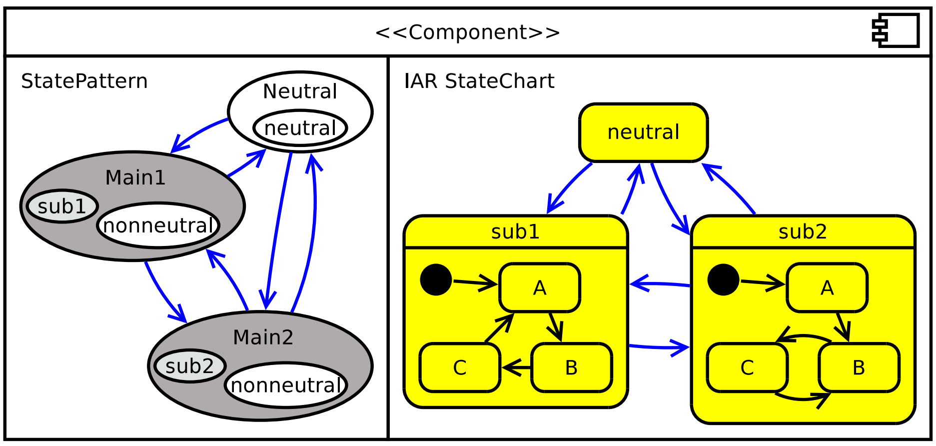
A StateChart inside of a SmartTasks using a substate

- The state chart from IAR Visual State is included into a SmartTask.
- Each SmartTask can include its own state chart.
- By deactivating a certain substate, the state-machine from a corresponding state-chart is frozen.
- This is reasonable for cases, where the state chart is only used for internal execution inside the task (e.g. a certain algorithm implemented as a state chart).

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Internal structure of a component in SMARTSOFT
SmartTasks in SMARTSOFT
Generic state automaton

Integration of a StateChart (2nd case with lose coupling)

Connecting substates with states from a StateChart



Lotz, Steck, Schlegel
A Generic State Automaton for a Component

Integration of a StateChart (2nd case with lose coupling)

Connecting substates with states from a StateChart

- Again, the state machine is included in a SmartTask, however a substate is not used this time.
- Each state-change (indicated in a state-change-handler of the state pattern) is forwarded to the state machine of the state chart.
- Thereby, the substates must be mapped onto corresponding events from the state chart and must be pushed onto the Event Queue.
- The state chart individually reacts on these events.

Example

Behavior coordination implemented as a StateChart.

Outline

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT

Parameter Port
Status Port

Parameter Port in SMARTSOFT
Overview

- A parameter port provides a generic way to send strategies, parameters, configuration and commands to a component.
- Therefore, component specific communication objects are used.
  - Specific setter and getter methods allow for appropriate individual checks of the syntax.
  - The parameter communication objects provide setter methods for strings, based on a Lisp like syntax (see examples in the following)
- This allows for automatic filling of the complex composite data types.

Lotz, Steck, Schlegel — A Generic State Automaton for a Component

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT

Parameter Port
Status Port

Parameter Port in SMARTSOFT
Examples

Example for CDL parameters

- Set the permitted velocity range [-200 800] mm/s
  `transvel(-200)(800)`
- Set the CDL strategy to drive reactive / to follow a person
  `strategy(reactive)`
  `strategy(followMe)`

Example for Planner parameters

- Delete all currently parametrized destination goals
  `deletegoal`
- Add new circled goal regions
  `setdestinationcircle(2000)(1000)(200)`
  x = 2000 mm; y = 1000 mm; Radius = 200 mm

Lotz, Steck, Schlegel — A Generic State Automaton for a Component

Parameter Port in SMARTSOFT

Further examples

Example for Speech parameters

- Change the grammar of the speech recognition
  `setgrammer("followMe.grxml")`

Example for Mapper parameters

- Set the region of interest in the current map
  `currparameter(10000)(20000)(-2000)(3000)`
  xSize = 10000 mm; ySize = 20000 mm;
  xOffset = -2000 mm; yOffset = 3000;

Outline

Status Port in SMARTSOFT

Collect and Observe states in single components

- A status port provides a generic interface to access data for introspection
- States can be used in monitoring components to administrate and observe a running system
- Further details are part of the masters thesis from Alex Lotz (coming soon)

Introduction
Concept of a generic state automaton in SMARTSOFT
Parameter- and Status-Port in SMARTSOFT
Parameter Port
Status Port

Bibliography

N. Ando, T. Suehiro, K. Kitagaki, and T. Kotoku.
Composite component framework for RT-middleware (robot technology middleware).
Proceedings, 2005 IEEE/ASME International Conference on Advanced Intelligent Mechatronics., pages 1330–1335, 2005.

Christian Schlegel.
Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach.
Phd, Ulm, 2004.

Lotz, Steck, Schlegel A Generic State Automaton for a Component

Bibliography

[1] Christian Schlegel. *Navigation and execution for mobile robots in dynamic environments: An integrated approach.* Phd thesis, University of Ulm, 2004.

[2] Christian Schlegel and Alex Lotz. ACE/SmartSoft – Technical Details and Internals. Technical Report 2010/01, University of Applied Sciences Ulm, Oktober 2010.

[3] Christian Schlegel, Andreas Steck, and Alex Lotz. Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model. In *Introduction to Modern Robotics*, chapter 28. iConcept Press Ltd, 2011. (coming soon).

Berichte des ZAFH Servicerobotik
ISSN 1868-3452

Herausgeber:
ZAFH Servicerobotik
Hochschule Ulm
D-89075 Ulm

http://www.zafh-servicerobotik.de/

Investition in Ihre Zukunft
gefördert durch die Europäische Union Europäischer Fonds
für regionale Entwicklung
und das Land
Baden-Württemberg