aimpulse intelligent systems

Aimpulse Spectrum
Aimpulse Implementation Specification

Dr. Jan D. Gehrke
Dr. Arne Schuldt

Aimpulse
Intelligent Systems GmbH
Fahrenheitstraße 1
28359 Bremen

Aimpulse Spectrum
Aimpulse Implementation Specification

Contents

Version: 1.3.2

1 Introduction

Aimpulse Spectrum is a Java-based runtime environment for multiagent systems. It is designed to execute a large number of agents in parallel. In addition, Spectrum eases agent development by means of agent behavior and interaction templates.

Aimpulse Spectrum supports agent communication following the messaging standards of the IEEE Foundation for Intelligent Physical Agents (FIPA). In contrast to other FIPA-compatible systems, Aimpulse Spectrum is not considered as a dedicated FIPA agent platform. Instead, it is designed as a more general environment for executing and evaluating parallel interacting processes. As a consequence, Aimpulse Spectrum does not support FIPA specialities such as distributed agent systems or agent mobility.

Aimpulse Spectrum plays to its strengths when it handles thousands or even a million agents at a time. Furthermore, it allows executing agents with the same implementation in very different execution modes. Besides the usual operation mode using real-time and real-world, the platform features simulation capabilities. To this end, Aimpulse Spectrum also considers simulation quality criteria (Schuldt, Gehrke, & Werner, 2008) that are usually not covered by general agent development frameworks. These criteria include time model adequacy, causality, and reproducibility.

1.1 Implementation

The diagrams depicted in this document are based on UML (Booch, Rumbaugh, & Jacobson, 2005), the Unified Modeling Language, as well as AUML (Odell, Parunak, & Bauer, 2000), the agent extension to UML. In addition, a full specification of the Application Programming Interface (API) is available on the Internet at developer.aimpulse.com.

The complete Java implementation of Aimpulse Spectrum has been programmed in a test-driven way with tests written first (Freeman & Pryce, 2009). Manifold JUnit-based unit tests further specify the behavior of the implementation.

The implementation of Aimpulse Spectrum follows the fail-fast principle (Shore, 2004). That is, any failures or illegal states are immediately reported at the interface of the system. Programmers using Spectrum are thereby supported to find and correct bugs easily already during development.

To enable the fail-fast technique already on the Java compiler level, Aimpulse Spectrum additionally uses specific types (Freeman & Pryce, 2009, p. 59). For instance, the language and encoding components of messages are represented as `LanguageIdentifier` and `Encoding-Identifier` instead of simply using the `String` class.

1.2 Structure of this Document

The general overview on the core platform is given in Chapter 2. It presents the overall architecture and how it can be employed in order to execute software agents. Furthermore, it describes the platform components and their interrelationship as well as the platform execution modes.

The controller is responsible for executing the software agents that populate the platform. Chapter 3 turns the attention to the actual agent implementation. To this end, it describes the

framework that enables structuring agent behavior more fine grained. In particular, agent behavior can be defined in sequence, in parallel, or in accordance with some finite-state machine.

Social ability is an important property of intelligent software agents. Therefore, software agents seldomly act in isolation. Chapter 4 describes the API for exchanging messages with other software agents. All messages are structured in accordance with the Agent Communication Language issued by the FIPA.

Message exchange in multiagent systems is often further structured by means of interaction protocols. Interaction protocols ease message handling by specifying the allowed flow of messages. Popular examples are the request and the contract net interaction protocols that have been standardised by the FIPA. Chapter 5 describes the behavior-based implementation of these protocols.

Multiagent systems are open systems. Agents may join and leave the system during runtime. Therefore, an agent does not necessarily know all of its interaction partners in advance. Chapter 6 describes the directory service with which agents can register themselves in order to be found by others.

In multiagent systems, there is often a potential for cooperation if a team can jointly achieve a goal that one agent cannot achieve in isolation. If agents have identified such a potential, they need a mechanism for team formation. Chapter 7 describes the behavior-based implementation of the Aimpulse team formation interaction protocol.

Aimpulse Spectrum can be used as a library that is instantiated by other software systems. Alternatively, it can be used as a stand-alone solution, e.g., for simulation. Chapter 8 describes how the platform and its agent can be configured and how such configurations can be read from and written to XML files.

2 Aimpulse Spectrum

Aimpulse Spectrum is a Java-based runtime environment for multiagent systems that is designed to execute a large number of software agents in parallel. This chapter presents the overall architecture and how it can be employed to execute agents.

Section 2.1 introduces the fundamental concepts of Aimpulse Spectrum. Section 2.2 describes the platform from the high level of user interaction. Section 2.3 covers the detailed level of process control and scheduling. Finally, Section 2.4 analyses the runtime characteristics .

2.1 Fundamental Concepts

In order to understand how to use Aimpulse Spectrum, the first glance has to be thrown on the fundamental concepts. Spectrum is not solely an agent execution platform but also a tool for analyzing complex parallel processes. Thus, the platform provides very flexible configuration facilities enabling the user to choose the best platform setting for the intended usage. For instance, one might want to employ Aimpulse Spectrum as an execution engine for multiagent systems with operational responsibilities in a corporate context. But then, this real-world application also requires previous thorough testing of multiagent behaviors. This can be done by using Aimpulse Spectrum as a simulation platform that executes application scenarios in a virtualized environment. Thus, the platform has two basic execution modes: *Simulation* and *Operation*. The general idea is that the agents, i.e., agent logic and communication, remain the same while only the execution mode changes.

This flexibility is accomplished by uncoupling the agents from the actual system environment and providing a generalized runtime environment that can be exchanged depending on the intended mode of usage. In particular, this runtime environment governs the way time advances from an agent's perspective. This is crucial for simulation. It also defines when and how agents shall be executed, i.e., their scheduling on the operating system and CPU level. For instance, one might want agents to be dedicated threads or leight-weight autonomous processing tasks executed in a thread pool.

Thus, Aimpulse Spectrum provides the facilities to setup the platform for a specific runtime environment. This is done by either using pre-defined implementations shipped with Spectrum such as for Discrete-Event Simulation (DES), or even implementing own environments. The following section introduces the technical backgrounds for the runtime environment, the general platform API, and the agents executed on that platform.

2.2 High-Level Platform

The interface `com.aimpulse.spectrum.Platform` is the usual point of entry for all interaction with Aimpulse Spectrum (Figure 2.1). Interactions include runtime commands for setup, starting and stopping the platform, as well as methods for adding agents. Additionally, the interface allows platform status queries such as the platform state.

Corresponding interface functions are as follows:

- `setup(RuntimeEnvironment)` configures the platform for its respective application purpose (see Section 2.2.1).
- `getMode()` returns the `Platform.Mode` specified during platform setup.
- `start()` starts the configured platform in a dedicated thread. That is, the method returns right after platform start.

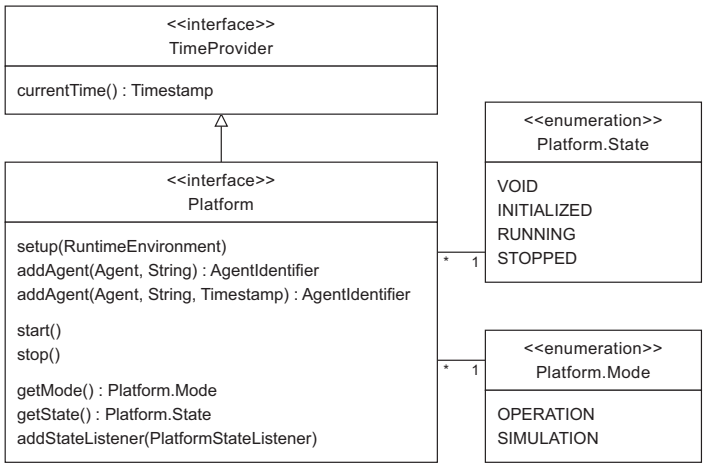

Figure 2.1: UML class diagram of the platform

- `stop()` causes the platform to stop as soon as possible, i.e., when all current agent computations are done. Thus, the method might return before the platform is actually stopped.
- `getState()` returns the current runtime state of the platform, e.g., `RUNNING`.
- `currentTime()` returns the current platform time (depending on `Platform.Mode`). The method is provided by extending the interface `TimeProvider`.
- `addStateListener(PlatformStateListener)` adds a state listener to the platform. Each new state is reported to all registered listeners.
- `addAgent(Agent, String, Timestamp)` adds an agent that is supposed to be executed on the platform. The methods takes the agent, the desired name and start time. On completed addition, the method returns a platform-unique `AgentIdentifier` based on the given agent name. The identifier serves as the agent's address in agent communication.
- `addAgent(Agent, String)` adds an agent that should be started as soon as possible. Depending on platform configuration, this is usually after all current agent computations are done.

The diagram in Figure 2.2 depicts the state sequence of the platform lifecycle.

2.2.1 Technical Platform Configuration

Before starting a `Platform`, it needs to be setup with the `RuntimeEnvironment` to be used. This technical platform configuration defines the `Platform.Mode`, i.e., `OPERATION` with real-time in a real environment or `SIMULATION` with logical time in a virtual environment, and its specific behavior with respect to runtime and process scheduling. Figure 2.3 shows the structure of the most important interfaces for technical platform configuration. The actual agent execution is handled by a `Controller` provided by the `RuntimeEnvironment`. The controller consults a `RuntimePolicy` for its lifecycle. This policy states when to stop agent execution, e.g., whether the platform should stop when any agent causes an uncaught exception. Each `RuntimeEnvironment` has a specific `TimeManager` which governs advancement of the platform time based on agent requests and external events. This is of particular importance in `SIMULATION` mode.

While a `RuntimeEnvironment` allows for configuring a `Platform`, both types are just interfaces. There are implementations for specific purposes and, potentially, with pre-defined `Platform` configuration. As a reference implementation, Aimpulse Spectrum features the

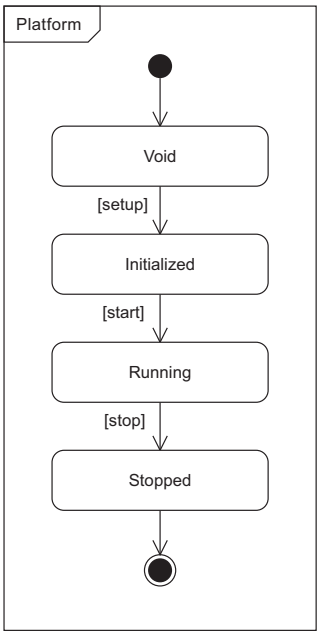

Figure 2.2: UML state machine of the platform

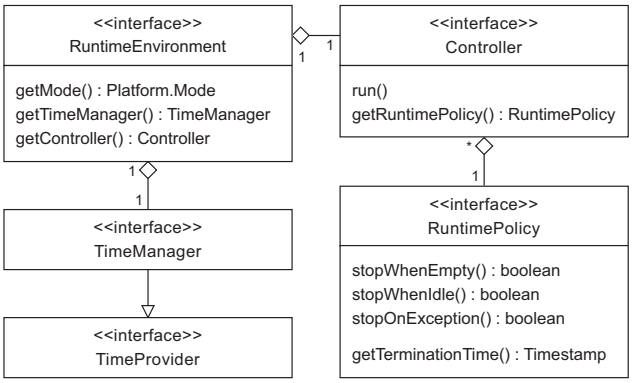

Figure 2.3: Overview UML class diagram of significant interfaces defining technical configuration

class `DefaultPlatform` without configuration. For DES, the `RuntimeEnvironment` class `DiscreteEventSimulation` is provided. `DiscreteEventSimulation` uses a discrete-event `TimeManager` that governs logical time by advancing it from event to next event. The configuration defines the start time used by the `TimeManager` and could also define a termination time in the applied `RuntimePolicy`.

Another `RuntimeEnvironment` would be applied for the `OPERATION` mode. Here, the `TimeManager` actually uses the system clock, no artificial time advancement is necessary. Note that the executed agents remain the same. No code adaptations are required.

Platforms can be conveniently configured by means of the `ConfigurablePlatform` class. Chapter 8 describes how the runtime environment and process definitions can be read from and written to a XML files.

2.2.2 Agents

Aimpulse Spectrum is a platform for executing agents and other concurrent processes. The class structure for agents is depicted in Figure 2.4. `Agent` is an abstract class. It implements the two interfaces `Process` and `Identifiable` with the generic type attribute `AgentIdentifier`. That is, an `Agent` instance is a process with a specific identifier. This unique identifier is the basis for addressing it in agent communication, and fulfills the requirements for FIPA agent identifiers. It is set by the `Platform` based on the agent name given when adding the agent.

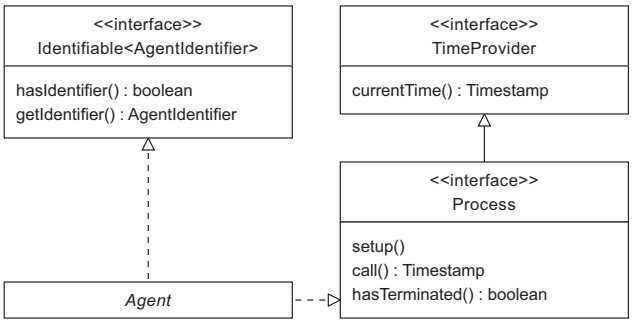

Figure 2.4: UML class diagram for class `Agent`

`Process` is the general interface for all processes executed by Aimpulse Spectrum. `Process` enables to query the current time by extending the `TimeProvider` interface. The semantics of the provided methods are as follows:

- `setup()` initializes the agent right before it is executed the first time. This is different from an object constructor because the agent is instantiated before it is added to the platform.
- `call()` executes the actual agent until it returns with a result. The result is a `Timestamp` that states when the agent requests to be activated again. If the agent has no specific request but wants to wait passively until some event (e.g., a message) occurs, it returns the maximum timestamp (`Timestamp.MAX`).
- `hasTerminated()` indicates whether the agent has terminated and does not want to be activated again. This property is evaluated by the platform after each agent call. Once `true`, the agent is removed from the platform.
- `currentTime()` returns the current time for this process. Note that, the process time does not have to match the current system time, especially when running a simulation. Thus, it is strictly discouraged to access system time via `System.currentTimeMillis()` in the Java API.

For agents that are supposed to act as communicating FIPA agents Aimpulse Spectrum uses class `CommunicativeAgent`. `CommunicativeAgent` provides messaging capabilities by implementing the `MessagingClient` interface. The usage of messaging functions is described in Chapter 4.

2.2.3 Events

Events are the means for interaction with processes running on the platform. For instance, agents can interact with each other by exchanging messages. When an agent sends a message, the platform generates an `Event` instance containing that message. Then, the platform takes

care of delivering that message to the addressees. When a message event or any other event is delivered, each receiver agent is activated even if there was no specific request of the agent to be activated at that time. Since Aimpulse Spectrum also serves as a simulation platform, the agent is activated at the (logical) time of the event. The platform also guarantees that events are processed in reproducible timestamp order.

2.3 Controller

The `Controller` interface is used for the internal management of agent execution. This includes the determination of agents that need execution (logical scheduling) as well as actual execution (physical scheduling). Thus, logical scheduling is about *who* needs execution and physical scheduling is about *how* this should be done.

The actual scheduling policy of the controller depends on its implementation. Aimpulse Spectrum provides a `SynchronizedController`. In this implementation, all agents with the same time for logical scheduling are executed within the same *tick* (or *cycle*). An agent to be activated at a later time is not scheduled before all agents of previous ticks are finished with their cycle. In physical scheduling however, the agents running within one tick might be executed in any order, sequentially or in parallel.

The class diagramm for `Controller` and `SynchronizedController` is depicted in Figure 2.5. Regarding the interface, `Controller` provides similar functions as `Platform`. But note that a platform might run multiple controllers at the same time or in sequence. Anyhow, `DefaultPlatform` has exactly one controller. The interface semantics for `SynchronizedController` are as follows:

- `addAgent(Agent, Timestamp)` adds a new agent (with `AgentIdentifier` already set) to the controller. The timestamp indicates the desired start time of the agent.
- `addEvent(Event)` adds a new event to the platform (in particular resulting from agent messages).
- `getRuntimePolicy()` returns the `RuntimePolicy` applied by this controller.
- `run()` runs the controller. This method is called upon `Platform.start()`.
- `stop()` causes this controller to stop execution of added processes. The method might return before the controller is actually stopped.
- `getState()` returns the current `Controller.State`. Is distinguishes whether the controller is `VOID` (not started), `RUNNING`, or `STOPPED`.
- `tick()` The sequential tick number of the current cycle. The first cycle has tick number 1. In `VOID` state the method returns 0.
- `currentTime()` The (logical) controller time for the current tick as returned by the applied `TimeManager`.
- `getStopOccasion()` returns the `StopOccasion` of the controller. It enables to query the reason of controller termination (e.g., the controller was idle).

2.3.1 Controller Execution Process

The diagram in Figure 2.6 shows the internal execution states of a `SynchronizedController`. These internal states differ from the exposed controller states. The controller state `RUNNING` is separated into the internal states *Evaluate* and *Execute*. The controller is in state *Void* before it is started by the platform. Usually, the platform runs the controller in a dedicated thread. When running, the controller first switches to state *Evaluate*. In this state `Synchro-`

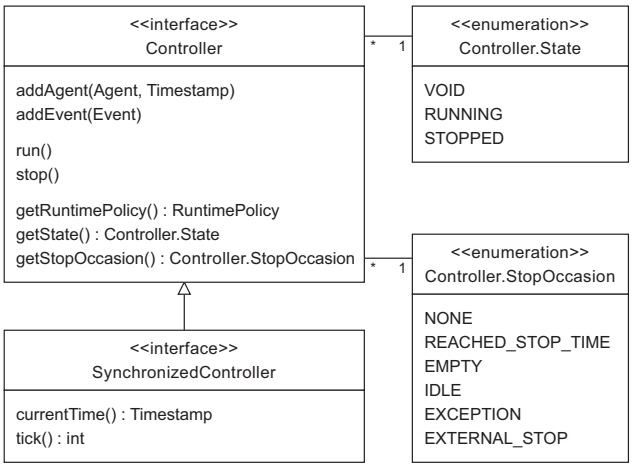

Figure 2.5: UML class diagram of the (synchronized) controller

`nizedController` adds newly added processes and determines the logical tick time of the next cycle. This is done by evaluating the lowest timestamp from intrinsic requests of agents (the agent start time in the first cycle) and extrinsic requests from added events.

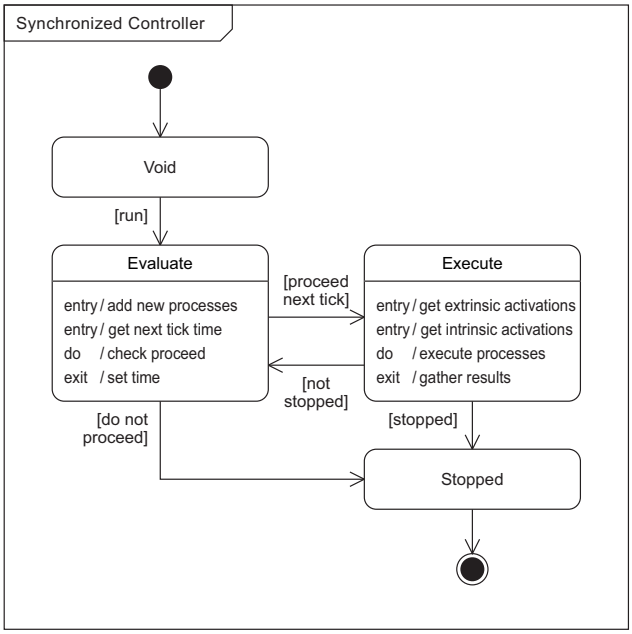

Figure 2.6: UML state machine of the synchronized controller

If the controller evaluates that it should proceed with the next cycle (depending on the applied `RuntimePolicy`) it sets the time to the lowest requested timestamp and switches to state *Execute*. In the other case, there is a reason for termination (e.g., a configured stop time is reached) and the controller switches to state *Stopped*. It is not possible to restart a controller once it stopped.

In state *Execute* the controller determines the agents that ought to be activated in the current cycle (i.e., logical scheduling). This is either due to intrisic or extrinsic requests. Then, all agents

to be activated are executed by handing them over to the applied *Scheduler* (physical scheduling). Finally, the controller waits for each agent to be finished and notes their result as a new intrinsic activation request.

If the platform operator triggered `stop()` for the controller, it directly switches to *Stopped* state after executing the current cycle. Otherwise, it continues by preparing the next cycle in state *Evaluate*.

2.3.2 Logical vs. Physical Scheduling

Logical scheduling is more or less the same for any application of the platform, no matter if it is operation or simulation. `Controller` manages the activation requests which are gathered from the result of each `Agent.call()` or might be triggered by extrinsic events (e.g., through delivered messages).

Physical scheduling might be very different depending on the current platform application and its implementation by the `Controller`. First, one must distinguish system time and logical time. While logical scheduling tells the logical time of desired agent execution, the actual execution with respect to system time might be very different. One reason is the runtime mode (`Platform.Mode`) of the platform. For simulation, system time and logical time usually do not correspond. Either the logical start time is different although time advancement is the same as for system time (as in real-time simulation) or time advancement differs. Differences in time advancement might be due to scaled real-time simulation or event-discrete simulation. Additionally, with parallel simulation, agents as logical simulation processes might even diverge in their logical time, called *local virtual time* (LVT). In the latter case, the agent controller has no definite current time. Instead, there is an interval from the earliest LVT to the latest LVT of the processes it handles.

2.4 Runtime Characteristics

Aimpulse Spectrum is designed to execute thousands or even more than a million agents concurrently. This is achieved by distributing agent execution to multiple threads and consequently multiple CPUs. In contrast to other systems, there is no fixed but an instant mapping between agent and thread. This way, the platform uses much less working memory and avoids disturbing CPU context switches that would be caused by uncoordinated thread scheduling by the operating system.

Besides the number of agents, the actual runtime performance of Aimpulse Spectrum largely depends on the frequency of agent activity and the system load caused by this activity. This is because the achievable parallelism is determined by these parameters. For the `SynchronizedController` the expected parallelism *EP*, i.e., processes running within one controler tick, is

$$EP = p \cdot \frac{tg}{\Delta a}$$

where p is the overall number of alive processes/agents on the platform, tg is the set minimum granularity for advancing time from tick to tick, and Δa is the average activation interval for a process. For instance, with $p = 10000$ agents, $tg = 1$ second of time advancement, and $\Delta a = 30$ minutes average activation the expected parallelism would be

$$EP = 10000 \cdot \frac{1}{30 \cdot 60} = 5.56$$

Consequently, in some ticks the parallelism might be higher or lower, e.g., only 2 agents per controller tick. In this case, for a hardware platform with 4 CPUs, actual CPU parallelism is expected to be significatly lower than 4. Expected and actual parallelism increase with more agents and a longer timespan for time granularity.

3 Event-Driven Agent Behavior

In Aimpulse Spectrum, the `Controller` is responsible for executing the software agents on the platform (Chapter 2). Every time an agent is executed, its `call()` method is called. The agent can then autonomously perform its computation. Afterwards, the agent specifies a `Timestamp` in order to request its next execution. This simple interface between the platform and the agent does not further restrict the way agents are implemented.

On the one hand, it is advantageous that agent implementation is rarely restricted. On the other hand, it is also a challenge for programmers to deal only with the raw interface. This particularly holds if an agent should be capable of complex behavior. Therefore, Aimpulse Spectrum provides a default implementation for event-driven agent behavior that eases agent implementation significantly.

Section 3.1 introduces the concept of event-driven agent behavior. Subsequently, Section 3.2 gives a broader overview on the event-driven agent behavior architecture with all related interfaces and classes. Finally, Sections 3.3 to 3.7 describe the predefined behavior implementations provided by Aimpulse Spectrum.

3.1 Event-Driven Behavior Concept

For rather simple agents, it suffices to implement their behavior directly in the respective `call()` method. The object-oriented programming paradigm allows structuring the implementation. The event-driven behavior paradigm helps structuring agent behavior even further. It allows grouping the implementation into states of agent behavior. Transitions between these states depend on the result of the preceding states. The respective state is kept even over multiple calls of the `call()` method. Furthermore, the execution of agent behavior is driven by events. This means that an agent behavior is only executed if a defined precondition holds.

The atomic entities in this paradigm are so-called behavior instances (Figure 3.1) which can be added to behavior-based agents.

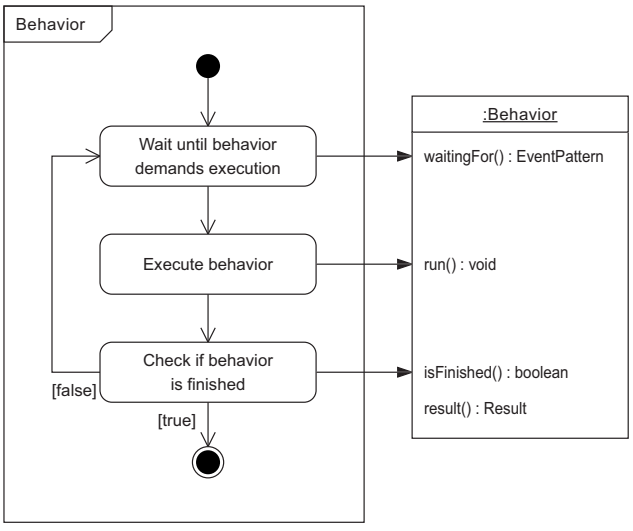

Figure 3.1: UML state machine of the event-driven agent behavior paradigm

Before a behavior is executed, it is checked when the behavior actually demands execution. To this end, the scheduler calls the `waitingFor()` method which returns an `EventPattern` object. The event pattern specifies a timeout `Timestamp` at which the behavior wants to be executed at the latest. If a behavior instance intends to handle arbitrary events, the correct event pattern is `EventPattern.ANY`. In this particular event pattern, the value for the timeout is set to `Timestamp.MIN`, i.e., the behavior is immediately executed.

Event patterns may also contain a `MessagePattern` which specifies which incoming messages the behavior awaits (Chapter 4). If a behavior instance intends to handle arbitrary messages, the correct message pattern is `MessagePattern.ANY`. If the `EventPattern` awaits messages without a timeout, the default value for the timeout is `Timestamp.MAX`, i.e., no timeout is defined.

As soon as the event pattern evaluates to `true`, the `run()` of the behavior is called in order to execute it. Actual implementations may further subdivide this method, e.g., with methods being executed exactly once before or after the execution of the behavior (Section 3.3).

Subsequent to executing the `run()` method of the behavior, the `isFinished()` method is called in order to find out whether the behavior demands further execution. If the behavior is not yet finished, the execution cycle starts again with checking when the behavior demands execution again.

Once, a behavior `isFinished()`, it may return a `Result` if the `result()` method is called. By default, the result is `null`. Following the fail-fast principle, behavior implementations are encouraged to throw an `IllegalStateException` if `result()` is called before the behavior `isFinished()`.

The following design contract holds for event-driven behaviors:

1. It is assumed that each behavior instance wants to be executed at least once. Hence, the initial return value of `waitingFor()` may not be `null`. After each execution cycle, the `waitingFor()` method may only be called again if `isFinished()` evaluates to `false`. Otherwise, `waitingFor()` may return any value including `null`.
2. `isFinished()` is always called after executing a behavior, never before. Hence, the `isFinished()` method of behaviors that only demand one execution can be implemented by simply returning always `true`.
3. If a behavior instance specifies a timeout for execution, it is guaranteed that the behavior is not executed before this timeout (unless, for instance, the message pattern of the event pattern evaluates to `true` for any incoming message). However, it is not guaranteed that the behavior is executed exactly at the time of the timeout. There may be a certain delay which is amongst others caused by the scheduling of other concurrent behavior instances.

Note that the `waitingFor()`, `isFinished()`, and `result()` interface methods are getters for the respective values. This means that their implementations should not include code not related to returning the respective values. For instance, `result()` should not contain any code that is intended to be executed after the behavior is finished. Due to its nature of being a getter, `result()` offers information to other classes which may decide to request it or not. Usually, the method is never called if subsequent agent activity does not depend on the result. Furthermore, all getters may be called multiple times. This could also interfere with program logic not related to returning the respective value of the getter.

The behavior concept of Aimpulse Spectrum resembles the one of JADE, the Java Agent Development Framework (Bellifemine, Caire, & Greenwood, 2007). The major distinction between the behavior concepts of JADE and Aimpulse Spectrum is the explicit specification of the event patterns behaviors are waiting for. This separates the waiting-for checks from the rest of the implementation. In particular, this allows executing only those behaviors which wait for any of the incoming messages.

3.2 Behavior Architecture

The class architecture of the event-driven behavior paradigm is as follows (Figure 3.2). Each behavior instance is connected to one behavior controller interface. The `BehaviorController` interface is implemented by behavior-based agents. The interface allows

- adding and removing behaviors to and from the controller,
- getting the current time,
- accessing the current events of the agent,
- getting the own identifier of the agent, and
- sending and receiving message.

To this end, the `BehaviorController` extends the `TimeProvider`, `EventAccess`, and `MessagingClient` interfaces. Each `Behavior` has exactly one `BehaviorController` while each `BehaviorController` may control multiple `Behavior` instances.

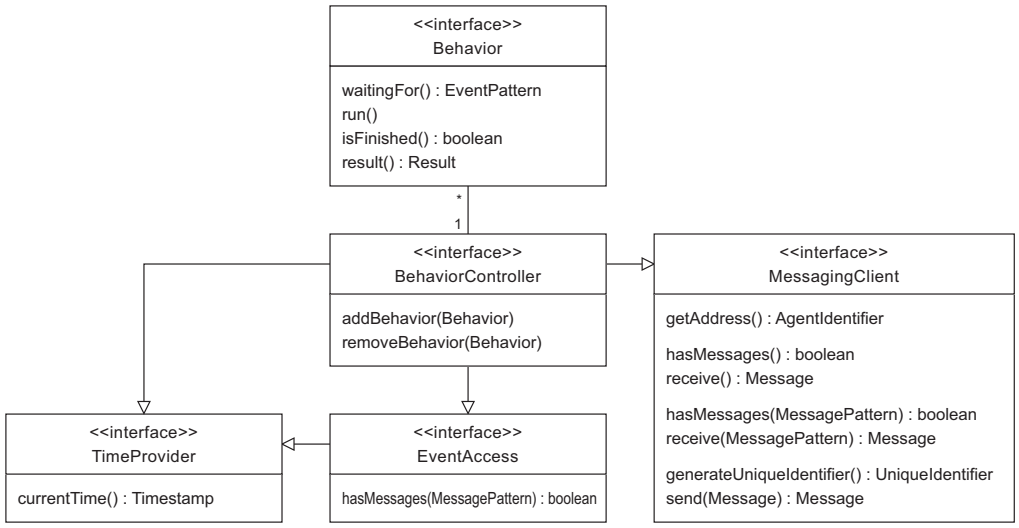

Figure 3.2: Behavior architecture

Spectrum provides the `BasicBehavior` as a default implementation for simple behaviors. Simple behaviors can be combined to complex ones with composite behaviors (Figure 3.3):

- `ParallelBehavior`
- `SequentialBehavior`
- `FiniteStateMachineBehavior`
- `FactoryBehavior`

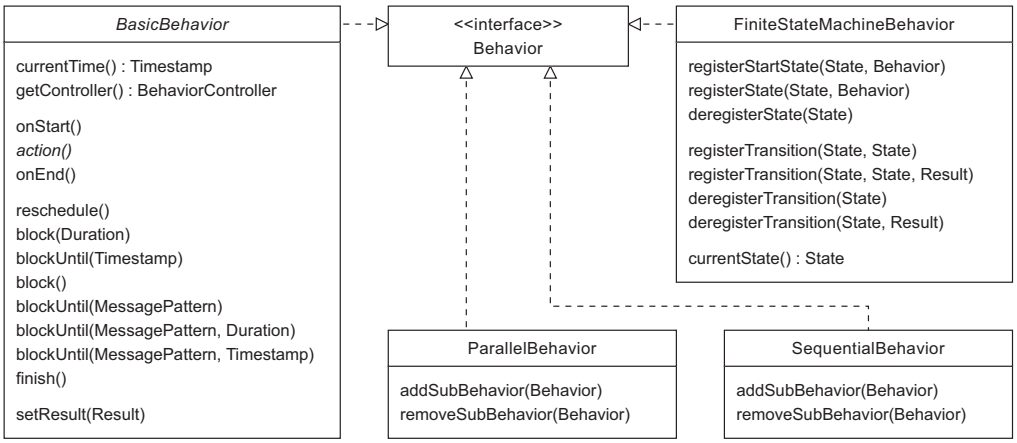

Figure 3.3: Behavior implementations

3.3 Basic Behavior

Instances of the `BasicBehavior` class are ready for execution immediately when they are added to a controller (Figure 3.4), i. e., their `waitingFor()` method returns the `EventPattern.ANY` pattern. The actual behavior of `BasicBehavior` instances can be implemented in the `onStart()`, `action()`, and `onEnd()` call-back methods. When a `BasicBehavior` is executed for the first time, the `onStart()` method is executed. Then, in every run (including the first run) the `action()` method is executed. After the final run of the `action()` method, the `onEnd()` method is called.

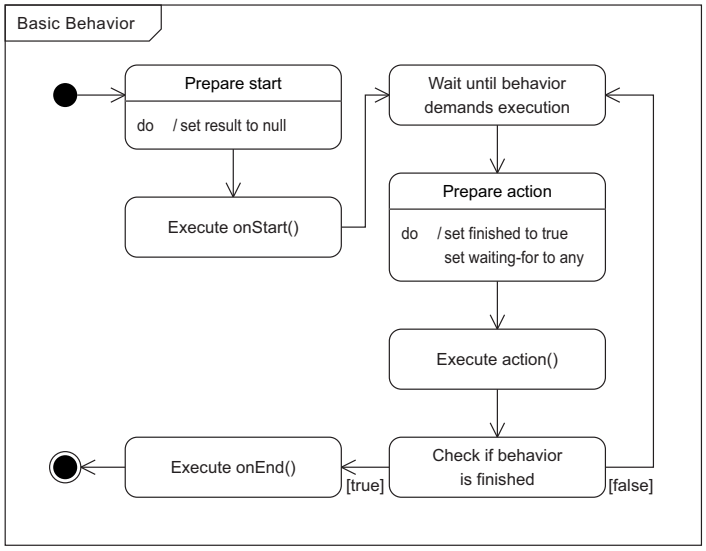

Figure 3.4: UML state machine of the basic behavior

Before each call to the `action()` method, the behavior is set to be finished in the prepare-action state. If the programmer does not explicitly request re-scheduling in the `action()` method, the behavior is executed only once and terminates afterwards. In order to request re-scheduling, the programmer can call one of the following methods (Figure 3.3):

- `reschedule()` to reschedule the behavior as early as possible.
- `block(Duration)` to block the behavior for the specified duration of time.
- `blockUntil(Timestamp)` to block the behavior until a specified point in time.
- `block()` to block the behavior until an arbitrary message arrives.
- `blockUntil(MessagePattern)` to block the behavior until a `Message` matching the `MessagePattern` arrives.
- `blockUntil(MessagePattern, Duration)` to block the behavior until a matching `Message` arrives but no longer than the specified duration of time.
- `blockUntil(MessagePattern, Timestamp)` to block the behavior until a matching `Message` arrives but no longer than the specified point in time.

In order to override a previous request for re-scheduling, the `finish()` method can be called. Note that these methods can be called only in the `onStart()` and `action()` call-back methods. Calling them in the `onStart()` method allows setting the event pattern for the first execution of the `action()` method. By contrast, these methods cannot be called in the `onEnd()` method because at that point the behavior is already finished. Following the fail-fast paradigm, illegal-state exceptions are thrown if the methods are called in disallowed states.

The result of the behavior can be set by means of the `setResult()` method. Note that the `setResult()` method can be called only in the `onStart()`, the `action()`, and the `onEnd()` call-back methods. Setting a result already in the `onStart()` method is useful whenever there is a default result which may be overriden later. Contrariwise, setting a result in the `onEnd()` method is useful if no result has been set before. Following the fail-fast paradigm, illegal-state exceptions are thrown if the method is called in disallowed states.

The `BasicBehavior` is prepared to be added again to a behavior controller after it is finished. This is possible because the prepare-action state does not only set the behavior to be finished, but also sets the waiting-for pattern to `EventPattern.ANY`. This means the behavior awaits immediate execution again when it is re-added to a behavior controller. Furthermore, the prepare-start state sets the result of a previous execution back to `null`. User-specific variables should be re-set manually in the `onEnd()` or `onStart()` methods.

3.4 Parallel Behavior

The `ParallelBehavior` is a composite behavior. All child behaviors that are added to this kind of behavior are executed in parallel, but not concurrently. The child behavior scheduling is as follows (Figure 3.5). The parallel behavior waits until any of its children demands execution. Then, it iterates over all children. Each child that demands execution is executed once. After executing a child, it is checked whether this particular child is finished. If it is finished, the child is removed from the internal scheduler of the parallel behavior. Afterwards, the next child demanding execution is called. After the parallel behavior has finished iterating over its children, it again waits until at least one child demands execution.

The children of this composite behavior can again be composite behaviors. The child behaviors are kept in an order-preserving set. The implementation of this set allows adding and removing child behaviors while iterating over the set.

The `EventPattern` a parallel behavior is `waitingFor()` is an aggregated event pattern of its children. If a parallel behavior has no child, its `EventPattern` is `null`. The timeout of the event pattern is the minimum timeout of all children. The aggregated message pattern is simply the universal message pattern. When a message arrives, it is checked for every child

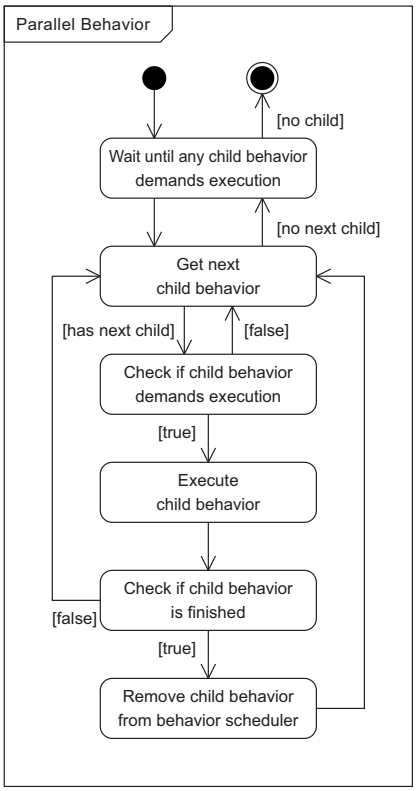

Figure 3.5: Child behavior scheduling of the parallel behavior

whether it awaits the message. This simplification is done for performance reasons. It means that it suffices to iterate once over the event patterns. The alternative would be to aggregate the message patterns. However, then the patterns had to be evaluated multiple times, once on every level of the behavior tree.

The `result()` of a parallel behavior is the result of its last child.

A parallel behavior is also the standard scheduler for behavior-based agents. That is, on the top level, each agent has a parallel behavior. Behaviors added to and removed from the agents are managed by this parallel behavior.

3.5 Sequential Behavior

The `SequentialBehavior` is a composite behavior. All child behaviors that are added to this kind of behavior are executed in sequence. The child behavior scheduling is as follows (Figure 3.6). A sequential behavior always only executes its first child. It waits until the first child demands execution. If the first child demands execution, it is actually executed. After executing a child, it is checked whether this particular child is finished. If it is not finished, the sequential behavior waits until the child again demands execution. If it is finished, the child is removed from the internal scheduler of the sequential behavior. The same procedure is repeated for the new first child until all children have been removed from the internal scheduler.

The children of this composite behavior can again be composite behaviors. The child behaviors are kept in an order-preserving set. The implementation of this set allows adding and removing child behaviors while iterating over the set.

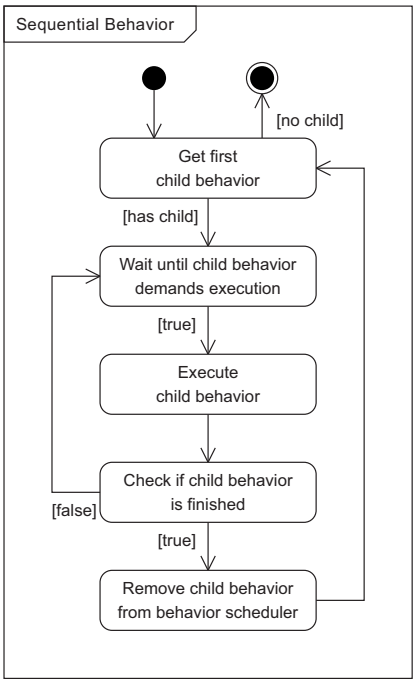

Figure 3.6: Child behavior scheduling of the sequential behavior

The `EventPattern` a sequential behavior is `waitingFor()` is the event pattern of the first child. If a sequential behavior has no child, its `EventPattern` is `null`. The `result()` of a sequential behavior is the result of its last child.

3.6 Finite-State Machine Behavior

The `FiniteStateMachineBehavior` is a composite behavior. Child behaviors can be registered as states of the underlying finite-state machine. Transitions between the states can be registered. Transitions may depend on conditions, namely the `result()` of the behavior in the origin state. The child behavior scheduling is as follows (Figure 3.7).

Each instance of the finite-state machine behavior starts with executing its start state. It waits until the start child demands execution. If the first child demands execution, it is actually executed. After executing a child, it is checked whether this particular child is finished. If it is not finished, the finite-state machine behavior waits until the child again demands execution. If it is finished, the next child is chosen based on the registered transitions. Finished children are not removed from their parent because there may be transitions calling them again. This procedure is repeated until a terminal state is reached.

The children of this composite behavior can again be composite behaviors. The implementation of the `FiniteStateMachineBehavior` allows adding and removing child behaviors while executing the finite-state machine behavior. However, the state that is currently executed cannot be removed.

The `EventPattern` a finite-state machine behavior is `waitingFor()` is the event pattern of its current state. If a finite-state machine behavior has no states, its `EventPattern` is `null`. The `result()` of a finite-state machine behavior is the result of its last state.

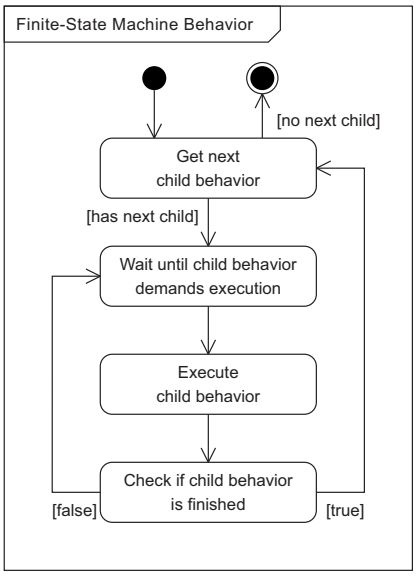

Figure 3.7: Child behavior scheduling of the finite-state machine behavior

3.7 Factory Behavior

Due to the nature of finite-state machines, some states might be never reached in some executions. However, some state behaviors may be quite heavy-weight, e.g., because they depend on database connections. Therefore, it is not reasonable to instantiate them already when the overall finite-state machine is instantiated. Instead, it is desirable to instantiate them dynamically on demand. Furthermore, in order to instantiate a behavior, input from a previous behavior might be required.

The `FactoryBehavior` implements lazy initialisation for behaviors (Figure 3.8). That is, the creation of a behavior is delayed until it is actually needed. When being executed for the first time, the `FactoryBehavior` instantiates an inner behavior by calling its `createBehavior()` method. If the inner behavior is `waitingFor()` any of the current events, it is immediately executed.

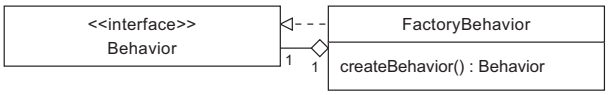

Figure 3.8: Factory behavior

After the instantiation of the inner behavior, the wrapping `FactoryBehavior` delegates all method calls to the created inner behavior. The only exception is the `isFinished()` method which is not delegated until the inner behavior has been executed for the first time. This ensures that the factory wrapper is not unscheduled if the inner behavior signalises that it is finished before the first run (which is acceptable because, according to the design contract for event-driven behaviors, `isFinished()` should only be checked after executing a behavior).

4 Agent Communication

Social ability is an important property of intelligent software agents. Therefore, software agents seldomly act in isolation. In Aimpulse Spectrum, software agents can exchange messages that follow the structure of the Agent Communication Language (ACL) specified by the Foundation for Intelligent Physical Agents (2002c). For performance reasons, the messages on Aimpulse Spectrum are directly exchanged as objects. That is, they are not transformed into an intermediary text representation.

Section 4.1 describes communicative acts and how they are represented in Aimpulse Spectrum. Subsequently, Sections 4.2 to 4.4 describe the structure of FIPA messages and how they can be created and exchanged. Finally, Section 4.5 explains how messages patterns can be defined in order to specify the messages a behavior is waiting for.

4.1 Communicative Acts

Communicative acts like `inform` or `request` indicate the intention of a message sent by a software agent. This is necessary in order to avoid ambiguities of the utterance (Huhns & Stephens, 1999, p. 87). The Foundation for Intelligent Physical Agents (2002d) has standardised a library of communicative acts. The purpose of this specification is to ensure interoperability between multiagent systems. The communicative acts are used in order to structure communication, e.g., in interaction protocols (Chapter 5).

Aimpulse Spectrum provides constants for all FIPA-defined performatives in the interface

`com.aimpulse.spectrum.mas.fipa.Performatives`

4.2 Messages

The structure of messages on Aimpulse Spectrum follows the standards of the Foundation for Intelligent Physical Agents (2002c). The Spectrum implementation distinguishes between the `Message` and `ModifiableMessage` interfaces (Figure 4.1).

The `Message` interface provides means to access all components of a message. The methods available cover access to the performative (type of communicative act), the participants in communication (sender, receivers, reply-tos), the content (either a `String` or a `Serializable` object), the content description (language, encoding, ontology), and conversation control (protocol, conversation-id, reply-with, in-reply-to, reply-by). In addition, it is possible to access additional user-defined parameters. Note that single-valued message components may be null, multi-valued message components may be empty collections. However, every message should at least have a performative specified.

The `ModifiableMessage` interface extends the `Message` interface. It provides additional means to set and unset the values of message components. Furthermore, multi-valued message components (receivers, reply-tos, user-defined parameters) can be reset completely by respective clear methods. Note that the sender cannot be set. This is to prevent fraud by spoofing the sender. The sender is set by Aimpulse Spectrum when a message is sent. It is also ensured that a sender that might have been set by an extended implementation of the `ModifiableMessage` interface is overwritten by Aimpulse Spectrum. Furthermore, the performative can only be replaced but cannot be unset. This is due to the fact that every message should at least have a performative.

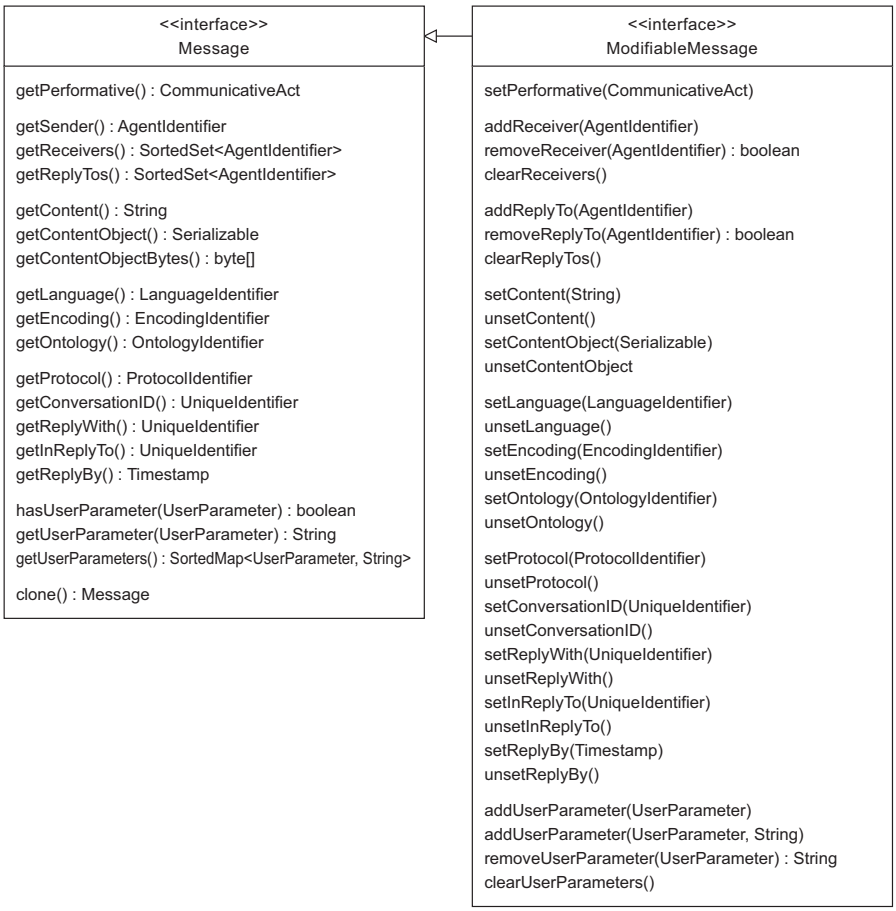

Figure 4.1: UML class diagram of the message interfaces

All messages implement the `toString()` method which can be used in order to print them for debugging. Note that messages are quite complex objects. Consequently, an implementation of the `toString()` method might also be computationally demanding. Furthermore, caching the string representation might not be advisable because it can be quite memory-consuming. Therefore, this method should better not be used excessively.

Currently, the `toString()` method of messages follows the specification for representing ACL messages in XML issued by the Foundation for Intelligent Physical Agents (2002b). The XML format has many advantages as there are manifold tools to process XML. For instance, XSLT (Kay, 2008) might be applied in order to create other representations (Foundation for Intelligent Physical Agents, 2002a) directly from ACL messages in the XML format.

However, developers should not rely on that output of the `toString()` method as it may be subject to change. Instead, they can employ the `XMLMessageWriter` which transforms messages into events for SAX, the Simple API for XML (McLaughlin, 2001, Chapters 3 and 4). These events can then be handled by an arbitrary SAX content handler for further processing including serialization. The other way round, the `XMLMessageHandler` can be employed in order to transform SAX events generated by an XML parser back into message objects.

4.3 Message Creation

Messages can be created conveniently by means of a `MessageBuilder` (Figure 4.2). The static `aMessage()` method creates a message builder with a specified communicative act. Alternatively, the static `aReply()` method creates a message (with a specified communicative act) that is a response to a former message. Components related to conversation control (protocol, conversation-id, reply-to, receivers) and content description (language and ontology) are then prepared automatically for the reply.

The message builder provides methods to specify messages conveniently in one line. The `build()` method finally creates a `ModifiableMessage`. This means that after building, all components can still be manipulated through the `ModifiableMessage` interface.

Unique identifiers are used in order to identify the conversation-id and the reply-with (and in-reply-to, respectively) message components. These identifiers should be unique throughout the whole multiagent system. The `generateUniqueIdentifier()` method of the `MessagingClient` interface provides such unique identifiers. It can be accessed by agent and behavior implementations by the extending `BehaviorController` interface (Chapter 3).

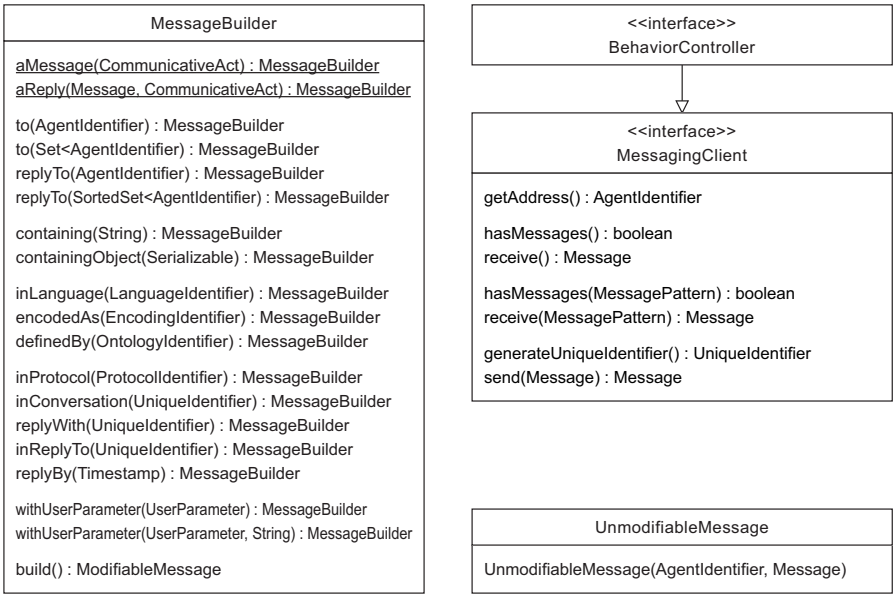

Figure 4.2: Classes related to message creation

4.4 Message Exchange

The `MessagingClient` interface also provides the means to exchange messages. The `hasMessage()` method checks whether the agent has incoming messages to be handled. The `receive()` method provides the next message from the internal message queue (and removes it from the queue). Methods with the same name but an additional `MessagePattern` parameter allow checking for and receiving specific messages.

Messages can be sent with the `send(Message)` method of the `MessagingClient` interface. The message transport system then creates an `UnmodifiableMessage` that also

includes the `AgentIdentifier` of the actual sender. Turning a `Message` into an `UnmodifiableMessage` ensures that the receiving agent cannot manipulate the message.

4.5 Message Patterns

Message exchange in agent-oriented programming is asynchronous. It differs from calling methods on objects synchronously in object-oriented programming. After sending a message, some time passes until the response from the correspondent arrives. This is, for instance, important in simulation where time advances only between agent executions. This means that a response can never arrive without time consumption during the same execution (Schuldt et al., 2008).

In practice, it is therefore usually the best choice to finish one execution cycle after sending a message. In order to be executed again when a response arrives, the behavior should specify a respective event pattern (Chapter 3) that covers the respective messages `MessagePattern` (Figure 4.3). Messages can be recognised, for instance, based on a conversation-id and and in-reply-to slots.

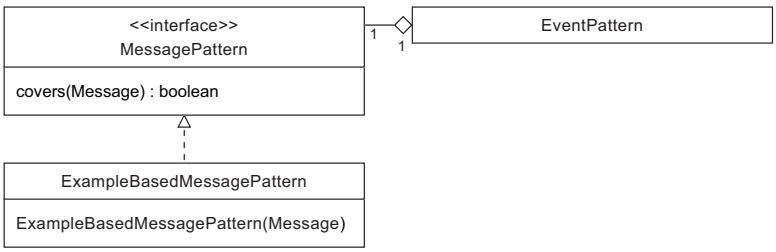

Figure 4.3: Message patterns

The `MessagePattern` interface contains the `covers(Message)` method in order to check whether a message is covered by the respective pattern. Implementations can check arbitrary components of the message. The `ExampleBasedMessagePattern` takes an existing message as an example template. The pattern covers a message if

1. For each single-valued component set in the template, the message has the same value like the template and
2. For each multi-valued component set in the template, the message has at least the same values like the template.

Note that programmers are relieved from dealing with message exchange and message patterns if they employ protocol implementations such as the FIPA Request or the FIPA Contract Net Interaction Protocols (Chapter 5). Therefore, it is advisable to employ such implementations whenever they are available.

5 FIPA Interaction Protocols

The standardized structure of messages helps agents understand and interpret them (Chapter 4). However, agent communication is usually not limited to sending single messages. Therefore, multiagent communication is often further structured by means of interaction protocols (Schuldt, 2011, Section 4.2.4). These interaction protocols ease message handling by specifying the expected and allowed flow of messages.

Section 5.1 introduces how interaction protocols can be implemented based on the behavior concept. Employing such predefined behaviors disburdens the programmer from the complexity of message processing. Instead, only call-back methods have to be implemented.

Section 5.2 describes the implementation of the request protocol which is probably one of the most popular protocols standardized by the Foundation for Intelligent Physical Agents (2002f). It allows an initiator to request some action from a responder. The focus of Section 5.3 is on the contract net, another prominent protocol issued by the Foundation for Intelligent Physical Agents (2002e). Compared to the request protocol, the contract net has an additional negotiation step before an action is actually performed.

5.1 Behavior-Based Implementation

The API of the protocol implementations for Aimpulse Spectrum resembles the one for JADE (Bellifemine et al., 2007). Like in JADE, agents for Aimpulse Spectrum can be implemented based on a behavior concept (Chapter 3). Therefore, programmers that are experienced with JADE can easily implement agents also for Aimpulse Spectrum. Note, however, that behaviors for Aimpulse Spectrum are based on an event-driven approach. Furthermore, while the API of the Spectrum behaviors resembles that of JADE behaviors, their internal implementation may differ significantly.

Each role a participant in the protocol can take, initiator and responder, is implemented by means of a respective behavior in Aimpulse Spectrum. Creating and handling messages is done by means of predefined call-back methods. Hence, it suffices to implement these methods. In particular, this means that programmers are disburdened from low-level message control.

The behaviors implementing the protocol roles are based on the finite-state machine behavior (Section 3.6). It is important to note that the standard implementation of this behavior executes each state only once per invocation of the run method. Hence, it requires at most n runs to receive n messages. It is thus ensured that CPU time is equitably shared by the agents even if they are executed in thread pool. Nevertheless, programmers do not have to consider rescheduling actively because this is done by the underlying implementation.

5.2 FIPA Request Interaction Protocol

The FIPA Request Interaction Protocol is defined as follows (Figure 5.1). The initiator sends a request to a group of responders. Each responder decides whether it agrees or refuses to perform the requested action. The initiator is notified about the decision. Likewise, a response is sent if the responder does not understand the original request. If the responder has performed the respective action, the initiator is informed about the result. Likewise, a failure message is sent if a failure occurred. Note that the agree message in the first step is optional. The FIPA standard mentions that it may be left out, for instance, if the result of the requested action is available very quickly.

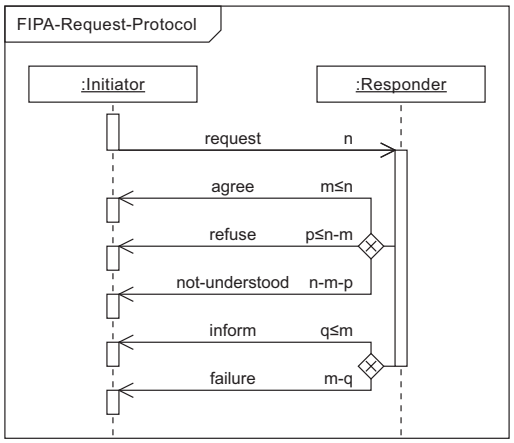

Figure 5.1: FIPA Request Interaction Protocol

A timeout can be defined by means of the `reply-by` slot of the initial `request` message. It refers to the time until the first response has to be sent. All `agree` messages that arrive after the timeout can most likely not be considered. If creating the result takes longer than the specified timeout, it suffices to send the `agree` message on time. The corresponding `inform` result notification can be sent even after the timeout is expired.

5.2.1 Behavior of the Request Initiator Role

The initiator role in the FIPA Request Interaction Protocol has been implemented by means of the finite-state machine behavior (Section 3.6). Note, however, that the initiator behavior does not inherit from the finite-state machine behavior but encapsulates it. The respective state machine is depicted in Figure 5.2.

The initiator behavior is initialized with a `request` message. This message can be further prepared in the first state of the state machine. The following send-requests state checks the prepared `request` message. If no message has been prepared or if the prepared message has no receivers, the state machine can terminate almost immediately. Before, however, the states handling all responses and result notifications, respectively, are processed. It is therewith ensured that these states are always activated independently from the actual number or responses and result notifications received.

If a message has been prepared successfully, it is provided with the `fipa-request` protocol identifier. Furthermore, a unique conversation identifier is set to the message in order to recognize replies. In order to distinguish replies by different responders, a cloned version of the original request is prepared for each receiver. Each message instance gets a unique `reply-with` identifier. Finally, a message pattern is created that waits for messages matching the correct `protocol`, `conversation-id`, and `reply-with`. If the original request contains a `reply-by` date, this timeout is additionally set for the behavior.

The following state receives replies to the request and delegates their handling to

- States handling responses (`agree`, `refuse`, `not-understood`).
- States handling result notifications (`inform`, `failure`).
- A state handling messages arriving out of the sequence of the protocol.

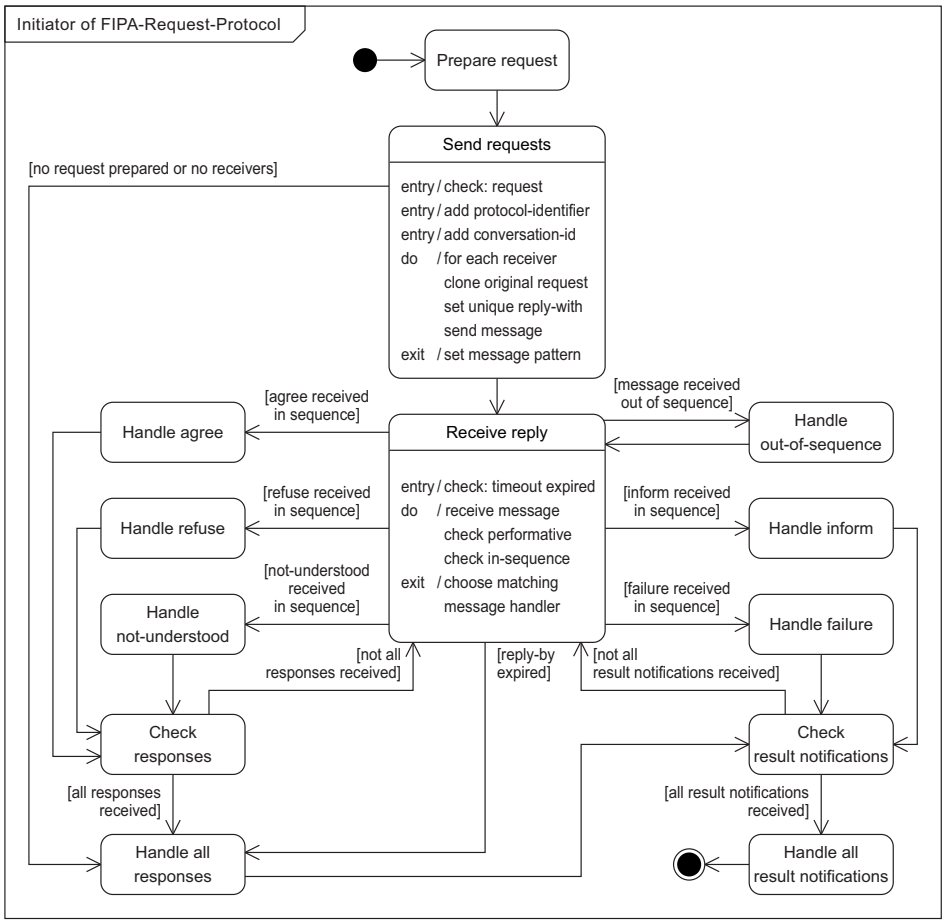

Figure 5.2: UML state machine of the request initiator role

After each time a response has been handled, another state checks whether all responses have been received already. If not all responses have been received yet, the next transition chooses the receive-reply state. Otherwise, a state that handles all response messages in total is activated. The same state is also called if the `reply-by` timeout expires (in this case, all unanswered sessions are removed). Subsequent to this state, the result notifications are checked. The same procedure applies after receiving each result notification. If not all result notifications have been received yet, the next transition chooses the receive-reply state. Otherwise, a state handling all result notification messages in total is activated. The result of this terminal state is also the result of the overall request initiator.

Behaviors exist for preparing requests and for handling replies (Figure 5.3). By default, each behavior implementation simply calls back a respective method on the initiator behavior. Hence, it suffices to implement the respective call-back method in an inherited class. As an alternative, however, it is possible to register a (simple or complex) behavior for each task.

All call-back methods have parameters through which they are provided with the required messages. If behaviors are registered instead, there is no general possibility to provide such parameters. Instead, there exists an interface that provides access to the required messages. It can be accessed with a getter method on the request initiator. Note that implementations

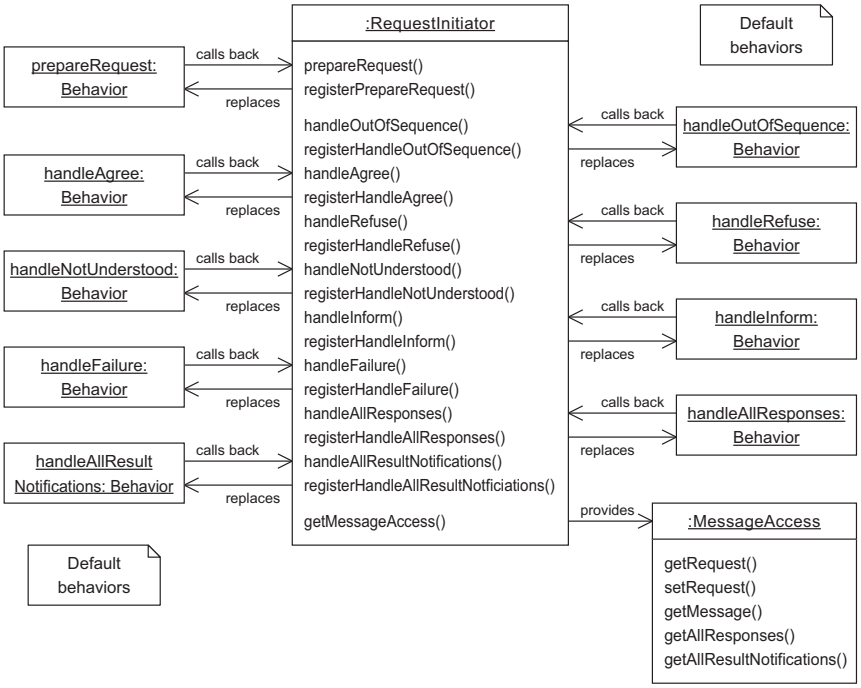

Figure 5.3: UML object diagram of the request initiator role

of this interface only provide access to the messages if their methods are used in the intended state of the finite-state machine. Otherwise, the methods will throw illegal-state exceptions.

5.2.2 Behavior of the Request Responder Role

The responder role in the FIPA Request Interaction Protocol has been implemented by means of the finite-state machine behavior (Section 3.6). Note, however, that the responder behavior does not inherit from the finite-state machine behavior but encapsulates it. The respective state machine is depicted in Figure 5.4.

The responder behavior waits for `request` messages with the `fipa-request` protocol identifier. It is possible to adjust the message pattern in the constructor. The entry point to the state machine is that a request is received. This request is handled in a respective state. The response prepared in this state is sent in the subsequent state. If a message with a `not-understood` or `refuse` performative has been prepared, the state machine returns to the first state and waits for new requests. If an `agree` message or no message has been prepared, a result notification is prepared in the following step. Also in this case, the state machine cyclically returns to the first state. This means that, after sending the result, it waits for receiving new requests.

Behaviors exist for handling requests and for preparing result notifications (Figure 5.5). By default, each behavior implementation simply calls back a respective method on the responder behavior. Hence, it suffices to implement the respective call-back method in an inherited class. As an alternative, however, it is possible to register a (simple or complex) behavior for each task.

All call-back methods have parameters through which they are provided with the required messages. If behaviors are registered instead, there is no general possibility to provide such parameters. Instead, there exists an interface that provides access to the required messages. It

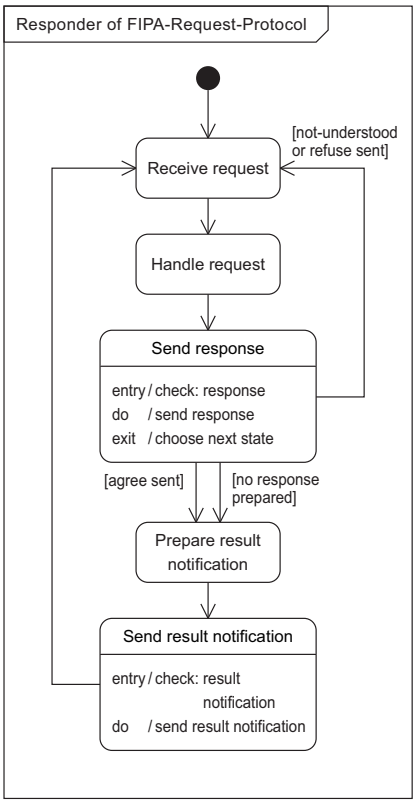

Figure 5.4: UML state machine of the request responder role

can be accessed with a getter method on the request responder. Note that implementations of this interface only provide access to the messages if their methods are used in the intended state of the finite-state machine. Otherwise, the methods will throw illegal-state exceptions.

In case that the call-back methods are used, it suffices to throw respective exceptions in case of a refuse, not-understood and failure. These exceptions are caught by the default behaviors and transformed into the expected reply messages.

5.3 FIPA Contract Net Interaction Protocol

The FIPA Contract Net Interaction Protocol is defined as follows (Figure 5.6). The initiator sends a call for proposals to a group of responders. Each responder decides whether it makes a proposal or refuses to perform the requested action. The initiator is notified about the decision. Likewise, a response is sent if the responder does not understand the original call for proposals. Out of all proposals, the initiator decides which proposals he accepts and rejects, respectively. The responder is is notified about the decision. If a responder has performed the respective action for an accepted proposal, the initiator is informed about the result. Likewise, a failure message is sent if a failure occured.

A timeout can be defined by means of the `reply-by` slot of the initial `cfp` message. It refers to the time until the first response has to be sent. All `propose` messages that arrive after the timeout can most likely not be considered. Note that the initial timeout does not cover the `inform` result notification for an accepted proposal.

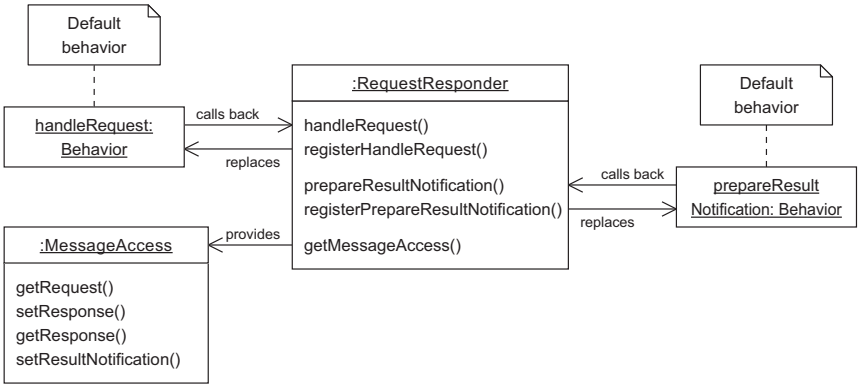

Figure 5.5: UML object diagram of the request responder role

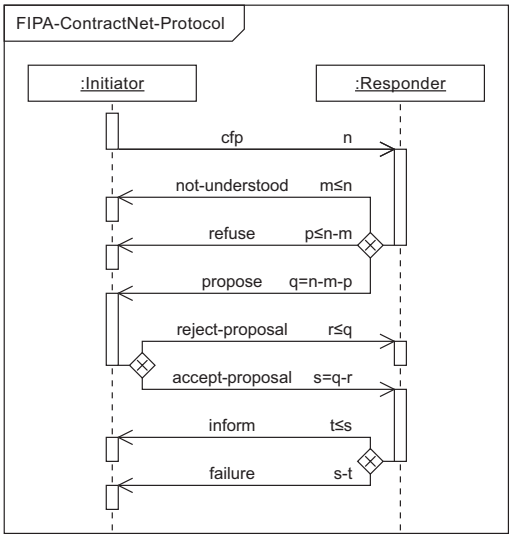

Figure 5.6: FIPA Contract Net Interaction Protocol

5.3.1 Behavior of the Contract Net Initiator Role

The initiator role in the FIPA Contract Net Interaction Protocol has been implemented by means of the finite-state machine behavior (Section 3.6). Note, however, that the initiator behavior does not inherit from the finite-state machine behavior but encapsulates it. The respective state machine is depicted in Figure 5.7.

The initiator behavior is initialized with a `cfp` message. This message can be further prepared in the first state of the state machine. The following send-call-for-proposals state checks the prepared `cfp` message. If no message has been prepared or if the prepared message has no receivers, the state machine can terminate almost immediately. Before, however, the states handling all responses and result notifications, respectively, are processed. It is therewith ensured that these states are always activated independently from the actual number or responses and result notifications received.

If a message has been prepared successfully, it is provided with the `fipa-contract-net` protocol identifier. Furthermore, a unique conversation identifier is set to the message in order to recognize replies. In order to distinguish replies by different responders, a cloned version

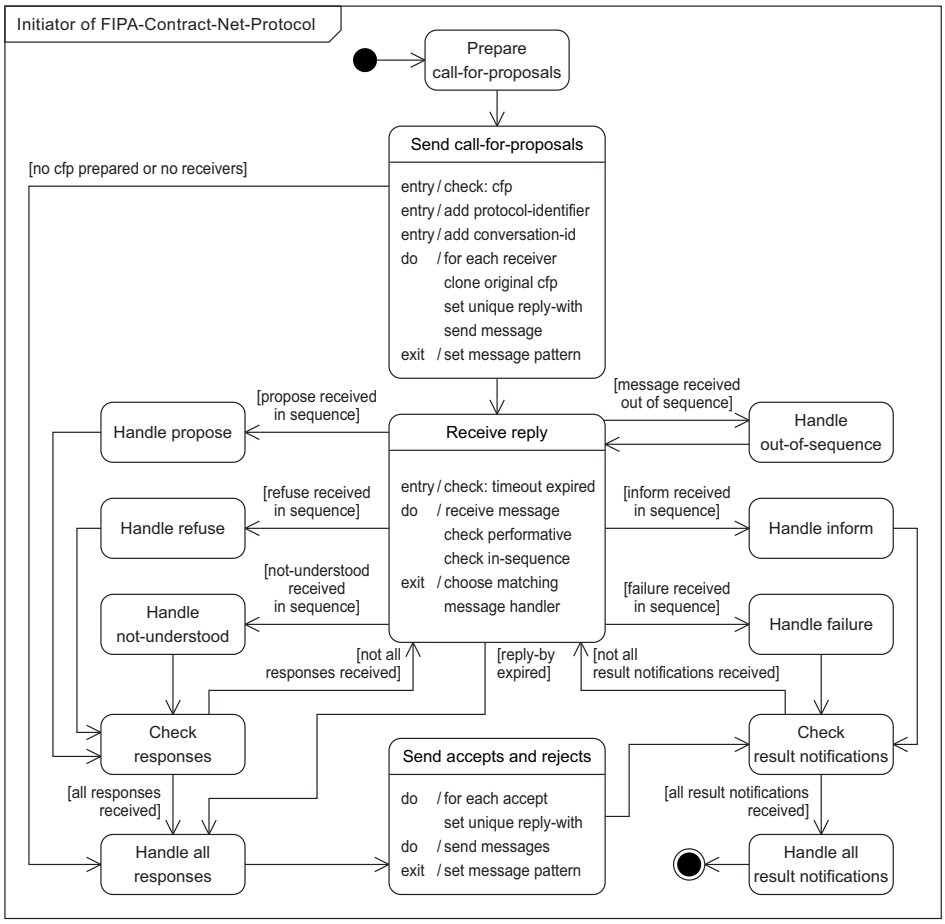

Figure 5.7: UML state machine of the contract net initiator role

of the original call for proposals is prepared for each receiver. Each message instance gets a unique `reply-with` identifier. Finally, a message pattern is created that waits for messages matching the correct `protocol`, `conversation-id`, and `reply-with`. If the original request contains a `reply-by` date, this timeout is additionally set for the behavior.

The following state receives replies to the call for proposals and delegates their handling to

- States handling responses (`propose`, `refuse`, `not-understood`).
- States handling result notifications (`inform`, `failure`).
- A state handling messages arriving out of the sequence of the protocol.

After each time a response has been handled, another state checks whether all responses have been received already. If not all responses have been received yet, the next transition chooses the receive-reply state. Otherwise, a state that handles all responses in total is activated. The same state is also called if the `reply-by` timeout expires (in this case, all unanswered sessions are removed).

A `propose` message can be either accepted or rejected in both the handle-propose state and the handle-all-responses state. To that end, a respective reply can be added to a collection of acceptances. By means of this collection, the respective handler can also see which other proposes the behavior has accepted or rejected already. Moreover, it can also modify previous

acceptances or rejections based on new information. This is possible because all acceptances are not sent until all responses have been received and handled.

Subsequent to handle-all-responses state, the result notifications are checked. The same procedure applies after receiving each result notification. If not all result notifications have been received yet, the next transition chooses the receive-reply state. Otherwise, a state handling all result notification messages in total is activated. The result of this terminal state is also the result of the overall contract net initiator.

Behaviors exist for preparing call for proposals and for handling replies (Figure 5.8). By default, each behavior implementation simply calls back a respective method on the initiator behavior. Hence, it suffices to implement the respective call-back method in an inherited class. As an alternative, however, it is possible to register a (simple or complex) behavior for each task.

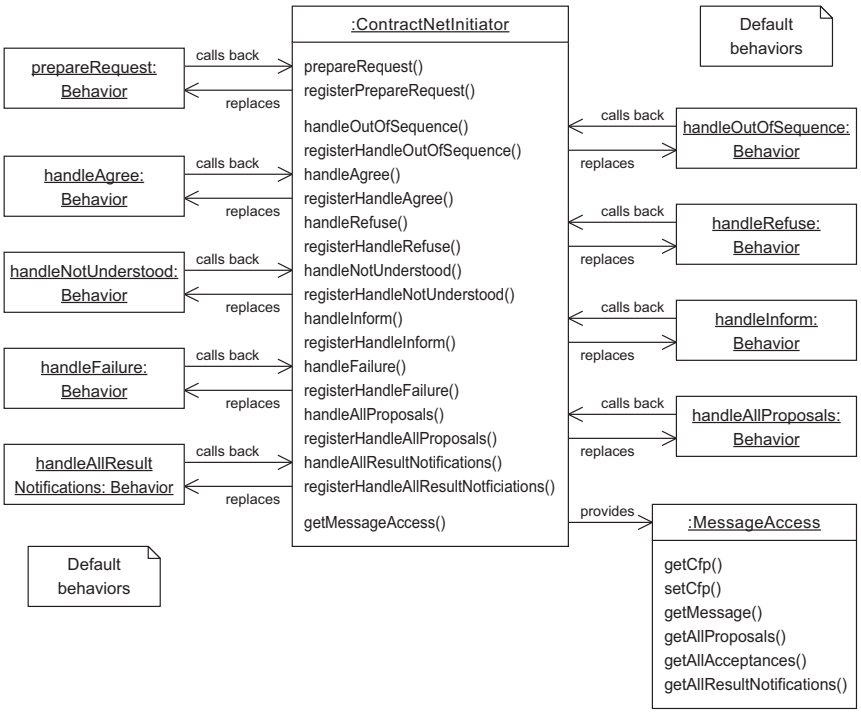

Figure 5.8: UML object diagram of the contract net initiator role

All call-back methods have parameters through which they are provided with the required messages. If behaviors are registered instead, there is no general possibility to provide such parameters. Instead, there exists an interface that provides access to the required messages. It can be accessed with a getter method on the contract net initiator. Note that implementations of this interface only provide access to the messages if their methods are used in the intended state of the finite-state machine. Otherwise, the methods will throw illegal-state exceptions.

5.3.2 Behavior of the Contract Net Responder Role

The responder role in the FIPA Contract Net Interaction Protocol has been implemented by means of the finite-state machine behavior (Section 3.6). Note, however, that the respon-

der behavior does not inherit from the finite-state machine behavior but encapsulates it. The respective state machine is depicted in Figure 5.9.

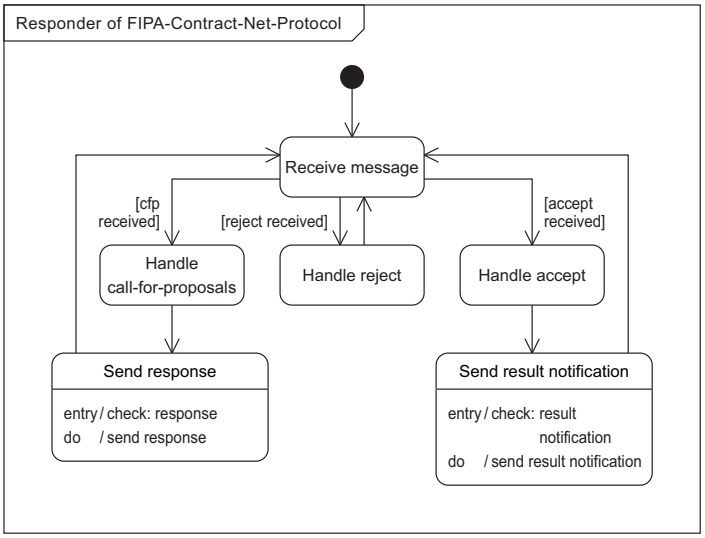

Figure 5.9: UML state machine of the contract net responder role

The responder waits for messages with the `fipa-contract-net` protocol identifier. It is possible to adjust the message pattern in the constructor. The entry point to the state machine is that a `cfp` message is received. The call for proposals is handled in a respective state. The response prepared in this state is sent in the subsequent state. If a message with a `not-understood` or `refuse` performative has been prepared, the state machine directly returns to the first state and waits for new messages. If a `propose` message has been prepared, a new session is instantiated internally in order to be able to react appropriately on `accept` and `reject` messages. Also in this case, the state machine cyclically returns to the first state. This means that, after sending the proposal, it waits for receiving new messages.

If a `reject` or an `accept` message is received, the respective session is retrieved. Subsequently, the corresponding state to handle the message is activated. The handle-reject state allows dealing with rejections. Afterwards, the state machine returns to the first state and waits for new messages. The handle-accept state allows preparing the result notification. Afterwards, the result notification is send to the initiator. Also in this case, the state machine is cyclic. This means that, after sending the result notification, it waits for receiving new messages.

Behaviors exist for handling messages (Figure 5.10). By default, each behavior implementation simply calls back a respective method on the responder behavior. Hence, it suffices to implement the respective call-back method in an inherited class. As an alternative, however, it is possible to register a (simple or complex) behavior for each task.

All call-back methods have parameters through which they are provided with the required messages. If behaviors are registered instead, there is no general possibility to provide such parameters. Instead, there exists an interface that provides access to the required messages. It can be accessed with a getter method on the contract net responder. Note that implementations

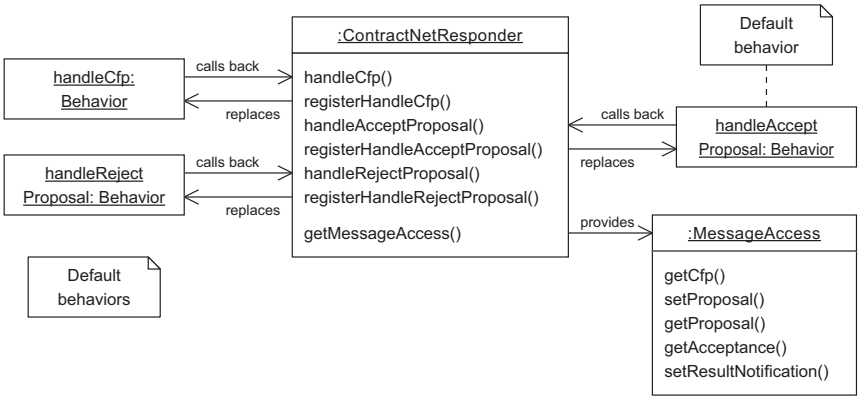

Figure 5.10: UML object diagram of the contract net responder role

of this interface only provide access to the messages if their methods are used in the intended state of the finite-state machine. Otherwise, the methods will throw illegal-state exceptions.

In case that the call-back methods are used, it suffices to throw respective exceptions in case of a refuse, not-understood and failure. These exceptions are caught by the default behaviors and transformed into the expected reply messages.

6 FIPA Directory Facilitator

Multiagent systems are open systems. Agents may join and leave the system during runtime. Therefore, an agent does not necessarily know all of its interaction partners in advance. The directory facilitator of FIPA multiagent systems provides the means to discover agents based on the services they provide (Foundation for Intelligent Physical Agents, 2004, Section 4.1). It is thus generally referred to as a yellow pages service.

Section 6.1 introduces the descriptions for agents and services. Section 6.2 describes the agent behaviors to register descriptions with the directory. Section 6.3 goes into more depth and describes the functions to manually update and search the directory. Finally, Section 6.4 describes the internal implementation of the directory facilitator and the management of its database.

6.1 Agent and Service Description

Registrations with the directory are based on agent and service descriptions (Figure 6.1). The `AgentDescription` interface provides access to the name of the agent as well as the supported protocols, ontologies, and languages (Foundation for Intelligent Physical Agents, 2004, Section 6.1.2). It is possible to specify the scope and the lease time of a description. Furthermore, the description contains a set of services provided by the respective agent.

In principle, all components of the agent description are optional. For all directory updates (register, deregister, or modify), however, an agent must at least specify its name. Otherwise, a failure exception occurs.

A `ServiceDescription` contains the name and the type of the service (Foundation for Intelligent Physical Agents, 2004, Section 6.1.3). It can also specify the supported protocols, ontologies, and languages of the service. Finally, also the ownership and special properties can be defined. Like for the agent description, all components of the service description are optional.

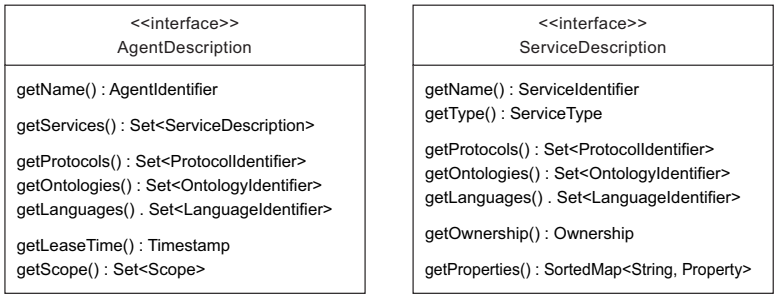

Figure 6.1: Agent and service description interfaces

Two classes exist in order to instantiate agent descriptions. The `AgentDescriptionBuilder` provides the means to conveniently create an agent description in one line of code (Figure 6.2 left). The static `anAgent()` method creates a builder. For every component of the description, there is a method to set its value. Note that single-valued components can only be set once. If all components have been specified, calling the `build()` method creates the actual description. Likewise, the `AgentDescriptionBuffer` can be employed for the stepwise creation of agent descriptions (Figure 6.2 right).

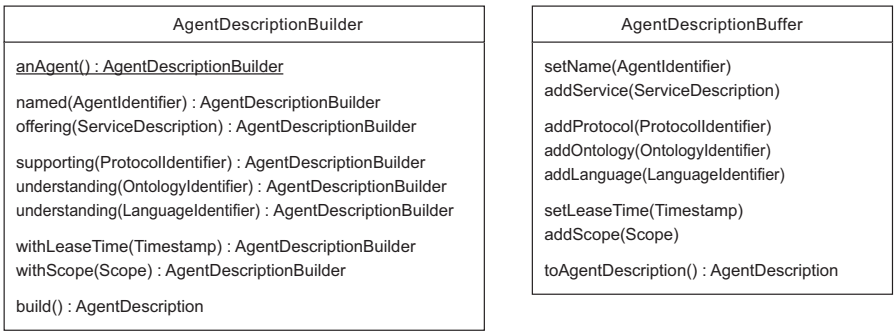

Figure 6.2: Classes related to agent description creation

Like for agent descriptions, there are also two classes to instantiate service descriptions. The `ServiceDescriptionBuilder` provides the means to conveniently create a service description in one line of code (Figure 6.3 left). The static `aService()` method creates a builder. For every component of the description, there is a method to set its value. Note that single-valued components can only be set once. If all components have been specified, calling the `build()` method creates the actual description. Likewise, the `ServiceDescriptionBuffer` can be employed for the stepwise creation of service descriptions (Figure 6.3 right).

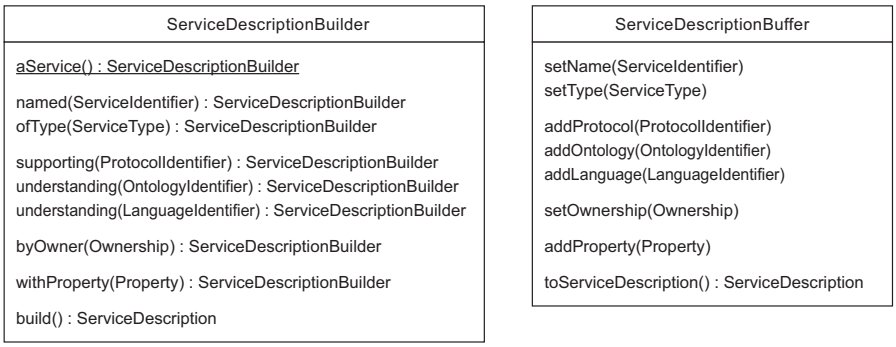

Figure 6.3: Classes related to service description creation

6.2 Behavior-Based Implementation

Four auxiliary behaviors (Figure 6.4) provide the most most convenient way for directory interaction: `RegisterInitiator`, `DeregisterInitiator`, `ModifyInitiator`, and `SearchInitiator`. They relieve programmers from dealing with low-level directory update and search functions. These behaviors are based on the FIPA Request Interaction Protocol `RequestInitiator` implementation (Section 5.2.1).

The behaviors can be combined easily with any other behavior. For instance, they can be added as states of a `FiniteStateMachineBehavior` with conditional transitions (Section 3.6). In that case, the succeeding behavior is not executed until the directory interaction is finished. The next behavior can then be chosen based on the result of the directory update or search. The behaviors provide call-back methods that can be implemented in order to handle successful and unsuccessful updates and search queries.

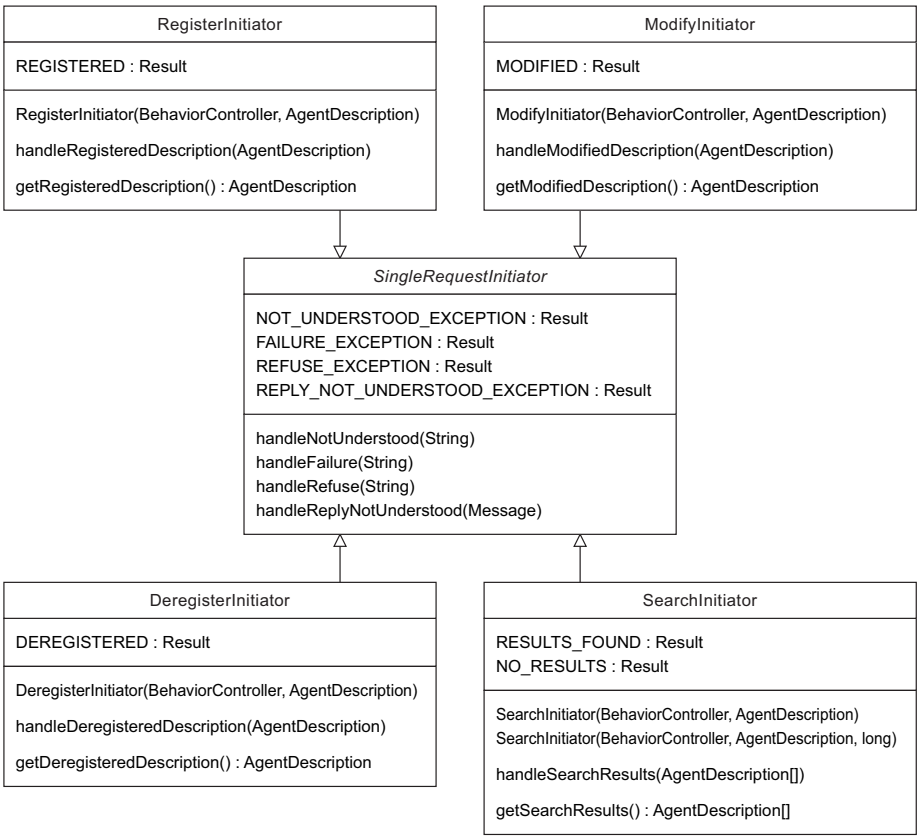

Figure 6.4: Agent behaviors for updating and searching the directory

Note that agents can only modify or deregister their registration if they are actually registered. Otherwise, a failure exception occurs. In contrast to the FIPA standard, the `SearchInitiator` does not restrict the maximum number of search results by default. However, they can be limited with an optional constructor parameter.

6.3 Update and Search Functions

Internally, so-called functions control the interaction with the directory. In order to update (register, deregister, or modify) a registration with the directory, an `UpdateFunction` is sent to the directory facilitator (Figure 6.5). This function contains the agent description and the type (register, deregister, or modify) of the update (Foundation for Intelligent Physical Agents, 2004, Sections 6.2.1 to 6.2.3).

If an update is successful, the directory returns the new directory record. This may distinguish from the registration request. In particular, the directory may remove the lease time if it does not support this feature (Foundation for Intelligent Physical Agents, 2004, Section 5.2.1).

In order to search the directory for registered agents, a `SearchFunction` is sent to the directory (Figure 6.6). The search function contains a template agent description and search constraints (Foundation for Intelligent Physical Agents, 2004, Section 6.2.4).

The `SearchConstraints` (Foundation for Intelligent Physical Agents, 2004, Section 6.1.4) allow specifying the maximum number of results as well as the maximum search depth (if federated directory facilitators are employed). A unique search identifier ensures that queries

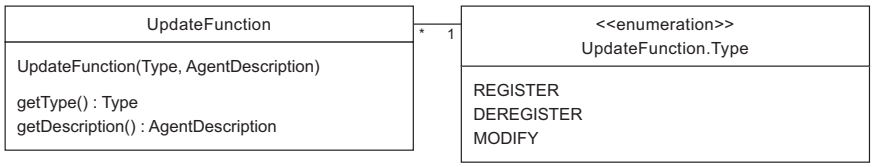

Figure 6.5: Directory update function

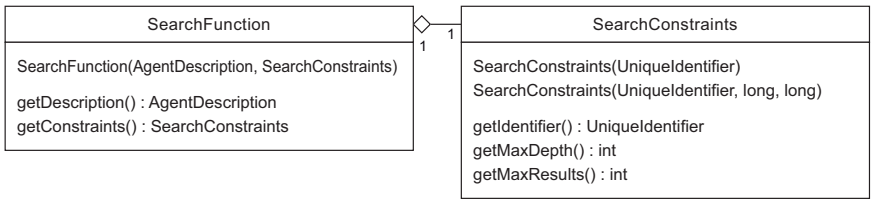

Figure 6.6: Directory search function

can be correctly handled with federated directory facilitators (Foundation for Intelligent Physical Agents, 2004, Section 6.1.3). Note that the current version of Aimpulse Spectrum does not support federated directory facilitators. Therefore, the max-depth parameter is currently not evaluated.

Following the standard of the Foundation for Intelligent Physical Agents (2004, Section 6.2.4.1), a directory entry matches a query if:

1. For each single-valued component set in the template, the entry has the same value like the template and
2. For each multi-valued component set in the template, the entry has at least the same values like the template.

6.4 Directory Management

The directory of Aimpulse Spectrum is administered by an agent. All tasks related to directory management are implemented in the `DirectoryManager` class which is a `Behavior` (Figure 6.7). The `FIPAPlatform` of Aimpulse Spectrum instantiates a directory agent and provides it with this behavior. The directory supports the FIPA Request Interaction Protocol. Therefore, the implementation of the `DirectoryManager` behavior is based on the `RequestResponder` (Section 5.2.2).

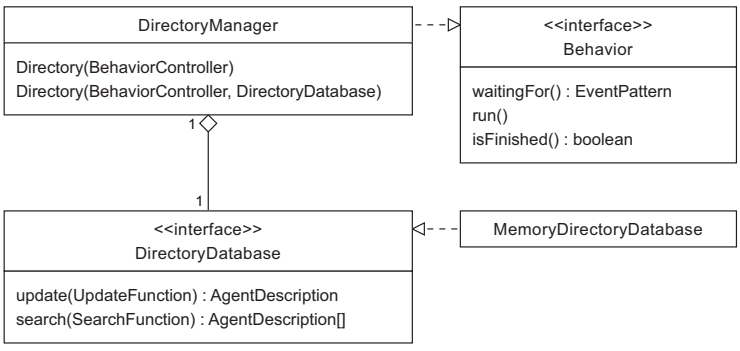

Figure 6.7: UML class diagram of the directory facilitator

All registered agent descriptions are stored in a `DirectoryDatabase`. The interface of this database provides the directory with the means to administer (update and search) its registrations. The default implementation is the `MemoryDirectoryDatabase` which holds all agent registrations as Java objects in memory. Other implementations of the interface might, for instance, employ an SQL database in order to be more scalable.

Note that the default `MemoryDirectoryDatabase` does not consider the lease time of agent descriptions. By contrast, when the directory notifies agents about their successful registration, they are informed that the registration time is not limited.

7 Agent Team Formation

In multiagent systems, there is often a potential for cooperation of agents. For instance, a team of agents might be able to jointly achieve a goal that one agent cannot achieve in isolation. Or, a team can achieve a goal better or more easily than one single agent. In order to be able to react flexibly on changing requirements or environments, agents should therefore be able to establish organisational structures like teams adaptively.

Different approaches exist for team formation (Schuldt, 2011, Chapter 6). A common approach is to apply the contract net interaction protocol for team formation (Section 5.3). An agent can initiate the contract net in order to announce missing capabilities required to solve its intended goal to other agents. Others can propose how they could support the initiator. Out of all proposals, the initiator can then form the best team for the intended task.

Another goal of team formation is to profit from economies of scale by using resources jointly. In that scenario, it might not be appropriate to have only one single point in time at which the team is formed. By contrast, it is desirable that additional participants can join the team later in order to use resources even more efficiently. This can be achieved by changing the initiator role of team formation from the team manager to the team member.

Section 7.1 introduces the Aimpulse interaction protocol for dynamic team formation. Subsequently, Section 7.2 describes the functions to match, join, and leave a team. Finally, Section 7.3 documents the behavior-based implementation of the agent roles in team formation.

7.1 Team Formation Interaction Protocol

The Aimpulse Team Formation Interaction Protocol (Schuldt, 2011, Chapter 6) allows partitioning agents into distinct teams. If the criterion for team formation is an equivalence relation, all members of one team have the same properties with respect to the relation. The resulting teams are dynamic in the sense that agents may join them later. The involved roles are the participant, the FIPA directory facilitator (Chapter 6), and the team manager. The protocol is defined as follows (Figure 7.1).

Agents may choose to form a team if they have identified a potential for cooperation (Schuldt, 2011, Chapter 5). If so, each instance of the team formation protocol is instantiated by the participant that intends to join a team. Initially the participant assumes that there is no team matching its requirements. Therefore, it registers itself as a team manager with the directory facilitator. Then, it searches the directory for all team managers for the respective purpose.

The participant contacts all team managers found and provides them with its properties in order to request a match. If another team manager has registered itself earlier for the same team, the participant deregisters itselfs and requests to join this team. Otherwise, it is actually a team manager itself.

7.2 Match, Join, and Leave Functions

In order to participate in team formation and to interact with their team managers, agents can employ three functions: `MatchFunction`, `JoinFunction`, and `LeaveFunction` (Figure 7.2). These functions are transmitted in messages to the respective team managers. Each function contains a `TeamDescription` which again contains the `AgentIdentifier` of the initiating agent as well as the `ServiceIdentifier` and the `ServiceType` of the team to

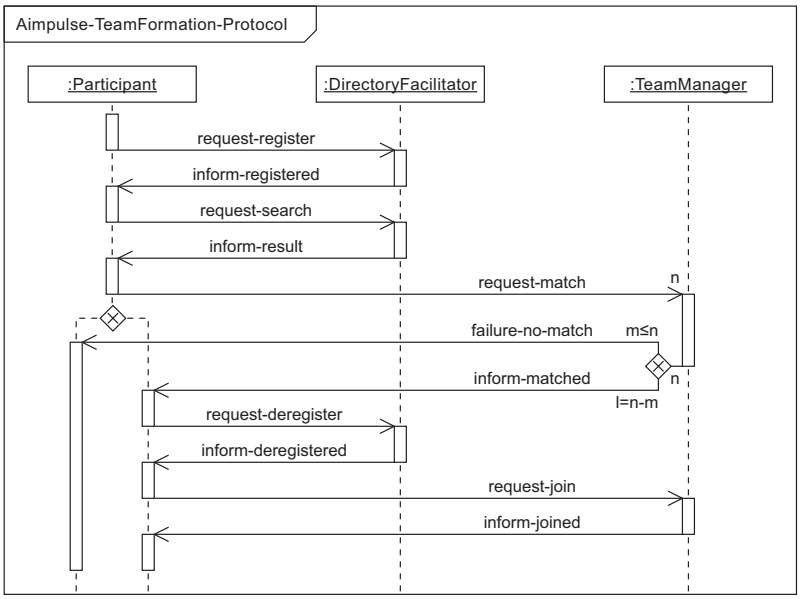

Figure 7.1: Aimpulse Team Formation Interaction Protocol

be established. The `TeamDescription` is an interface which can be extended with the actual components required by the intended application.

Only the `MatchFunction` informs about its result with a specific `MatchResult` reply. The `MatchResult` contains the `AgentIdentifier` of the matching team manager, the `double` distance, and the `Timestamp` at which the respective team has been established.

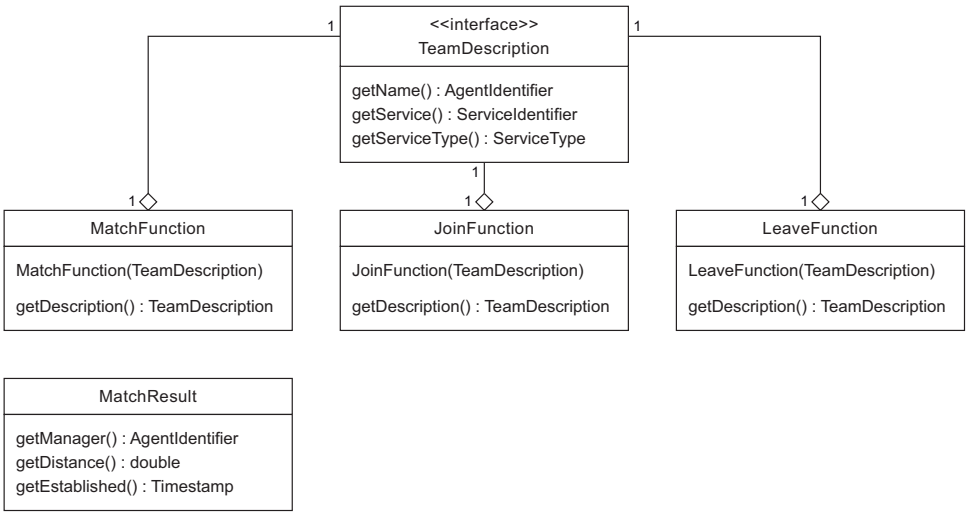

Figure 7.2: UML class diagram of the team functions

7.3 Behavior-Based Implementation

The Team Formation Interaction Protocol has been implemented based on event-driven agent behaviors (Chapter 3). Figure 7.3 gives an overview of the behaviors responsible for coordinating the interaction between participants in team formation and team managers.

The `TeamParticipant` behavior is implemented based on a `FiniteStateMachineBehavior`. States contained in the finite-state machine include a `MatchInitiator` and a `JoinInitiator`. Apart from that, also initiators communication with the directory are contained. The `LeaveInitiator` behavior is not part of the `TeamParticipant` behavior. This is due to the fact that is not part of team formation. However, team members may employ the `LeaveInitiator` behavior later in order to leave a team again.

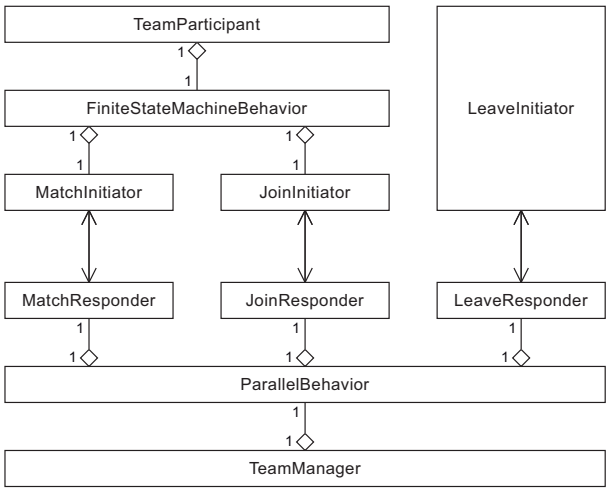

Figure 7.3: UML class diagram of the behavior architecture for team formation

The `TeamManager` behavior aggregates the `MatchResponder`, a `JoinResponder`, and a `LeaveResponder` behavior. To this end, it employs a `ParallelBehavior` which executes the member behaviors in parallel.

All initiators and responders are implemented based on the implementation of the FIPA Request Interaction Protocol (Section 5.2). Note, however, that they have their own protocol identifiers in order distinguish them more efficiently:

- `aimpulse-team-match`
- `aimpulse-team-join`
- `aimpulse-team-leave`

7.3.1 Behavior of the Team Participant Role

The participant role in team formation is implemented by the `TeamParticipant` behavior (Figure 7.4). Each `TeamParticipant` has a `TeamManagerFactory` instance in order to instantiate a team manager behavior if the participant itself is a team manager. The `DefaultTeamManagerFactory` is instantiated with a `TeamMatcher` and (optionally) a `TeamDatabase`. If no `TeamDatabase` is specified, the default `MemoryTeamDatabase` is used.

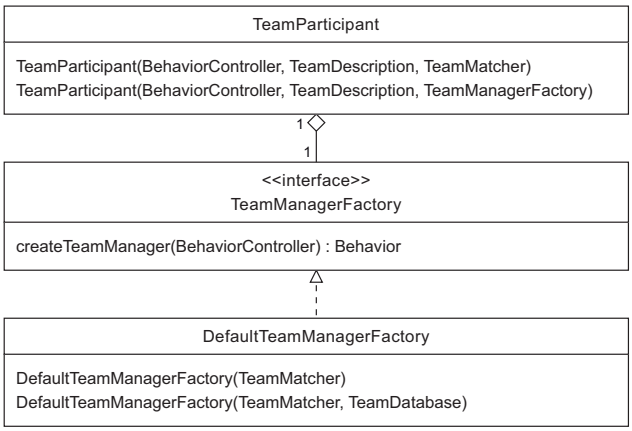

Figure 7.4: UML class diagram of the team participant role

Note that each `TeamManagerFactory` can create only one team manager instance. This is due to the fact that, otherwise, two team manager instances would use the same matcher and database. Cloning these dependencies, however, is impossible. Both interfaces might be implemented by the same class (e.g., in order to update the matcher reference based on the current database entries). If both dependencies were cloned, they would no longer be the same class.

The `TeamParticipant` can be instantiated with a `TeamManagerFactory`. Alternatively, it suffices to specify a `TeamMatcher`. In that case, a `DefaultTeamManagerFactory` with the `MemoryTeamDatabase` is used.

Initially, the `TeamParticipant` employs the `TeamManagerFactory` in order to instantiate and install a `Behavior` that acts as a team manager (Figure 7.5). Afterwards, it employs a `RegisterInitiator` behavior in order to register with the directory facilitator as a team manager (Chapter 6). The service identifier and the service type for registering with the directory are derived from the `TeamDescription`. Subsequently, it employs a `SearchInitiator` behavior in order to search the directory for registered team managers. Then, all team managers retrieved from the directory (apart from the agent itself) are contacted by means of the `MatchInitiator` behavior.

If no matching team manager is found, team formation terminates and the team participant actually remains a team manager. If a match is found, the participant checks whether the other manager is younger than itself. In that case, the other agent is expected to deregister and join. Otherwise, the participant chooses the team manager with the best match (i.e., the one with the smallest distance). It employs the `DeregisterInitiator` in order to deregister as a team manager from the directory. Afterwards, it employs the `JoinInitiator` in order to join the team manager chosen. Finally, it informs the `TeamManager` behavior about the dissolution and uninstalls it.

7.3.2 Behavior of the Team Manager Role

The `TeamManager` role in team formation is implemented by a `Behavior` (Figure 7.6). It has a `dissolve()` method by which the `TeamParticipant` can inform it about the dissolution of the team. In that case, a `TeamManager` could take the necessary steps such as

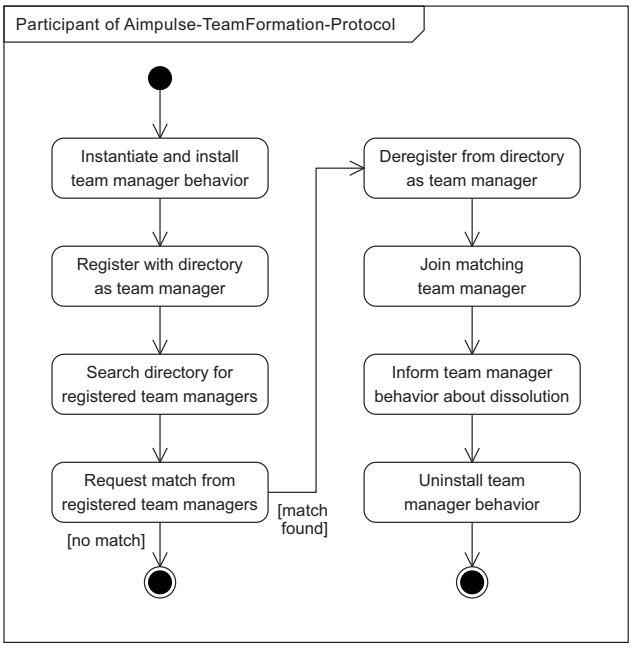

Figure 7.5: UML state diagram of the team participant role

informing its members. The `DefaultTeamManager` implementation does not do anything if its `dissolve()` method is called.

The `DefaultTeamManager` implementation aggregates a `MatchResponder`, a `JoinResponder`, and a `LeaveResponder` behavior in a `ParallelBehavior` (Figure 7.3). These behaviors, however, are only responsible for coordinating the interaction with team participants. All decisions regarding team formation are delegated to the `TeamMatcher` interface. The `TeamDatabase` administers team member registrations (Figure 7.6). The `MatchResponder` consults the `TeamMatcher` in order to decide whether a `TeamParticipant` matches the team. The `JoinResponder` again consults the `TeamMatcher` (to prevent fraud) and delegates join requests to the `TeamDatabase` afterwards. The `LeaveResponder` delegates leave request to the `TeamDatabase`.

The `TeamMatcher` interface provides the `distance()` methods which determines the distance of the `TeamDescription` of a `TeamParticipant` to the respective description of the whole team. The distance is represented by a `double` value with *0* denoting equality. If the distance is above a certain threshold (which depends on the domain of application), the `TeamMatcher` throws a `NoMatchException`. Consequently, the `TeamMatcher` sends a `Failure` message to the `TeamParticipant`.

The `TeamDatabase` specifies the interface for administering the team member registrations (Figure 7.6). The methods `handleJoin()` and `handleLeave()` are called by the `DefaultTeamManager` behavior in order to notify the database about changes. For both operations, the `TeamDatabase` may throw a `FailureException`. The `handleJoin()` method is supposed to throw an exception if the requesting participant is already member of the respective team. The `handleLeave()` method should throw an exception if the requesting participant is not member of the respective team.

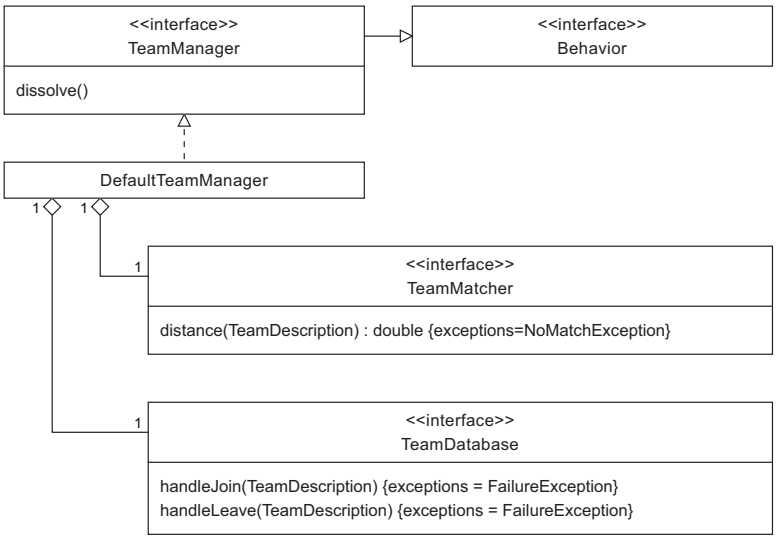

Figure 7.6: UML class diagram of the team manager role

There are currently two implementations of the `TeamDatabase` interface (Figure 7.7). The `MemoryTeamDatabase` keeps all registrations as Java objects in memory. For performance reasons, the `MemoryTeamDatabase` does not support registering listeners for database updates. However, if this feature is required, the `ListenableTeamDatabase` may be employed. The `ListenableTeamDatabase` implements the so-called decorator pattern (Gamma, Helm, Johnson, & Vlissides, 1995, p. 175) and can wrap any other `TeamDatabase` implementation, e.g., the `MemoryTeamDatabase`.

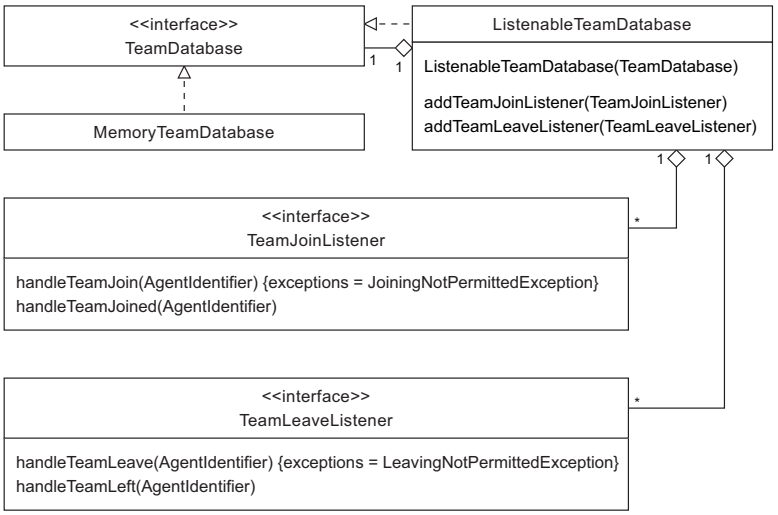

Figure 7.7: UML class diagram of the team database

The `ListenableTeamDatabase` allows registering instances of the `TeamJoinListener` and the `TeamLeaveListener`, respectively. The `handleTeamJoined()` and `handleTeamLeft()` methods are called if the respective operation is finished. However, it is also possible to check these operations in advance. The `handleTeamJoin()` method allows throwing a `JoiningNotPermittedException`. Likewise, the `handleTeamLeave()` can

throw a `LeavingNotPermittedException`. These methods can be used by components in order to influence the team member administration. For instance:

- Joining might not be permitted if the potential member is known to be malicious.
- Joining might not be permitted if a maximum number of team members is reached.
- Leaving might not be permitted if other contracts have to be dissolved first.

Note that listeners are only provided with the `AgentIdentifier` of the participant. They do not have access to the `TeamDescription` of the participant. This is due to the fact that joining and leaving should not be rejected based on the description. The respective checks should be left to the `TeamMatcher` which is consulted already before the `TeamDatabase` comes into play. If the `TeamMatcher` considers a `TeamParticipant` to be eligible for joining the team (based on the team description), the `TeamDescription` should not be a reason for later rejections.

Both, the `TeamMatcher` and `TeamDatabase` interfaces, can be implemented by the same class. This is useful, if the `TeamMatcher` depends on the current member registrations. Consider, for instance, a `distance()` function based on the means of the current member descriptions. As another example, the distance may depend on the nearest neighbour.

8 Configuration

Aimpulse Spectrum can be used as a stand-alone solution or as a library that is instantiated by other software systems. If it is used as a library, the other system will most likely itself configure the `Platform` and its agents. Particularly for simulation, however, it is useful to read scenario configurations from files to make them well-documented and reproducible.

Spectrum can be configured by means of XML files. Simply reading a configuration from an XML file already suffices for most applications. Sometimes, however, it advantageous to be able to further process the configuration. Therefore, the configuration architecture of Aimpulse Spectrum is also prepared to transform and to write configurations.

Section 8.1 describes the format of XML configuration files for Aimpulse Spectrum. Subsequently, Section 8.2 goes into more depth and introduces the overall configuration architecture. Section 8.3 addresses how platform configurations can be processed and persisted.

8.1 XML Configuration Files

Aimpulse Spectrum is shipped with scripts to start it with an XML configuration. On Microsoft Windows operating systems, Aimpulse Spectrum can be started as follows:

```
spectrum.bat -c scenario.xml -cp scenario.jar
```

For Unix-like operating systems, the respective start script is named `spectrum.sh`. The `-c` parameter allows specifying the XML configuration for the platform. The agent implementation can be added to the class path with the `-cp` parameter.

The XML configuration for Aimpulse Spectrum scenarios comprises two parts: the definition of the runtime environment and the processes to be executed (Listing 8.1).

Listing 8.1: XML configuration example

```
<?xml version="1.0" encoding="UTF-8"?>
<spectrum version="1">
  <scenario>
    <runtimeEnvironment
        implementation="com.aimpulse.spectrum.core.DiscreteEventSimulation">
      <attribute name="startTime">2011-09-01T08:30:00.000Z</attribute>
    </runtimeEnvironment>
    <processes>
      <process name="agent" implementation="com.example.staff.Employee"
          startTime="2012-09-15T09:15:00.000Z">
        <attribute name="name">Harold</attribute>
        <attribute name="salary">30000</attribute>
      </process>
    </processes>
  </scenario>
</spectrum>
```

The runtime environment is defined by means of the fully qualified name of a Java class that implements the `RuntimeEnvironment` interface (Section 2.2.1). The example in Listing 8.1 uses the default `DiscreteEventSimulation` execution mode of Aimpulse Spectrum. Attributes can be specified that are required by the chosen `RuntimeEnvironment`. In the

example, the start time of the platform is specified as an attribute. The runtime environment definition can be specified only once per configuration.

The `processes` section of the configuration specifies the processes to be executed, i.e., the agents (Section 2.2.2). The `implementation` of an agent is defined the fully qualified name of a Java class that implements the `CommunicativeAgent` interface. The `name` is used by the platform in order to generate a unique `AgentIdentifier` when the process is added to the platform. If the provided name is already used as an `AgentIdentifier`, the platform extends it with a serial number. Furthermore, it is possible to specify a `Timestamp` start time at which the respective process should be executed for the first time. If no start time is specified, the process is added immediately to the platform

Times are specified in the XML Schema `xs:dateTime` data type which is inspired by the ISO 8601 standard. Note that all times should include an explicit time zone in order to make them independent from interpretation, e.g., based on the current location of a user. The example uses the `Z` which stands for the UTC time zone.

8.2 Configuration Architecture

Internally, the Spectrum configuration API is stream-based. That is, an event is generated for each configuration element that is added to the configuration, e.g., when reading it from a file. The object can then be processed immediately. The advantage of the stream-based approach is that it is not necessary to keep the whole configuration in memory, possibly over multiple processing steps. Instead, its elements can be processed one after another.

The central interface for this purpose is the `PlatformConfiguration`. Its definition corresponds to the XML configuration. Like the XML configuration, also the `PlatformConfiguration` interface consists of two parts (Figure 8.1). On the one hand, the specification of the runtime environment (e.g., discrete event simulation). On the other hand, the specification of the simulation processes (e.g., agents).

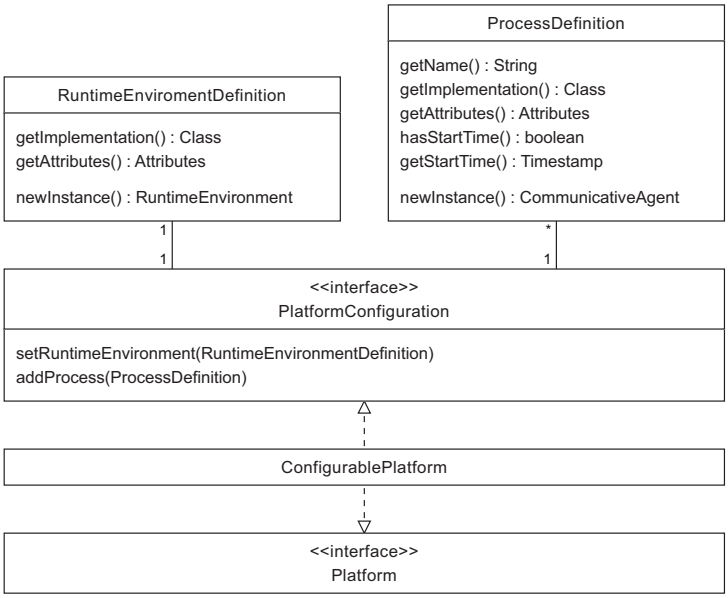

Figure 8.1: UML class diagram of the platform configuration

The `RuntimeEnvironmentDefinition` comprises a Java `Class` implementation and optional `Attribute` instances which further specify the implementation. The actual `RuntimeEnvironment` can be instantiated with the `newInstance()` method. The runtime environment definition can be specified only once per configuration.

A `ProcessDefinition` comprises a `String` name, a Java `Class` implementation, and optional `Attribute` instances which further specify the implementation. The name is used by the platform in order to generate a unique `AgentIdentifier` when the process is added to the platform. If the provided name is already used as an `AgentIdentifier`, the platform extends it with a serial number. Furthermore, it is possible to specify a `Timestamp` start time at which the respective process should be executed for the first time. If no start time is specified, the process is added immediately to the platform. Process definitions cannot be added to a configuration unless the runtime environment definition has been specified.

Both the `RuntimeEnvironmentDefinition` and `ProcessDefinition` classes are immutable (like the `java.lang.String`). Buffer and builder auxiliary classes help instantiate them conveniently.

A `RuntimeEnvironmentDefinition` can be created by means of the `RuntimeEnvironmentDefinitionBuffer` and `RuntimeEnvironmentDefinitionBuilder` auxiliary classes (Figure 8.2). A `RuntimeEnvironmentDefinitionBuffer` is instantiated with a Java `Class` implementation. Arbitrary `Attribute` instances can be added with the `addAttribute()` method. If the buffer is complete, the actual `RuntimeEnvironmentDefinition` can be created with the `toRuntimeEnvironmentDefinition()` method. The `RuntimeEnvironmentDefinitionBuilder` class provides the same functionality with a convenient one-line notation.

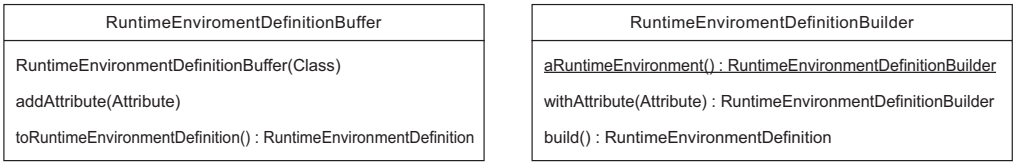

Figure 8.2: UML class diagram of the classes for runtime environment definition creation

A `ProcessDefinition` can be created by means of the `ProcessDefinitionBuffer` and `ProcessDefinitionBuilder` classes (Figure 8.3). A `ProcessDefinitionBuffer` is instantiated with a `String` name and a Java `Class` implementation. Arbitrary `Attribute` instances can be added with the `addAttribute()` method. Furthermore, it is possible to set the `Timestamp` start time with the `setStartTime()` method. If the buffer is complete, the actual `ProcessDefinition` can be created with the `toProcessDefinition()` method. The `ProcessDefinitionBuilder` class provides the same functionality with a convenient one-line notation.

8.3 Stream-Based Configuration Processing

The `ConfigurablePlatform` is a `Platform` (Section 2.2) implementation that also implements the `PlatformConfiguration` interface (Figure 8.1). Definitions of the runtime environment or processes are delegated directly to the underlying platform. Consequently, this type of platform can act as a consumer of a configuration read from a file.

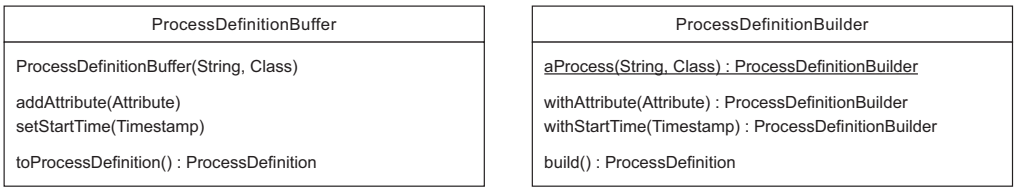

Figure 8.3: UML class diagram of the classes for process definition creation

Reading a configuration from an XML file can be accomplished with SAX, the Simple API for XML (McLaughlin, 2001, Chapters 3 and 4). The `XMLConfigurationHandler` implements the `ContentHandler` interface of SAX (Figure 8.4). Therefore, it can be registered as the content handler of a SAX `XMLReader`. The content handler transforms all XML events generated by the SAX parser into configuration elements. These configuration elements are then directly added to the `PlatformConfiguration` specified in the constructor of the `XMLConfigurationHandler`.

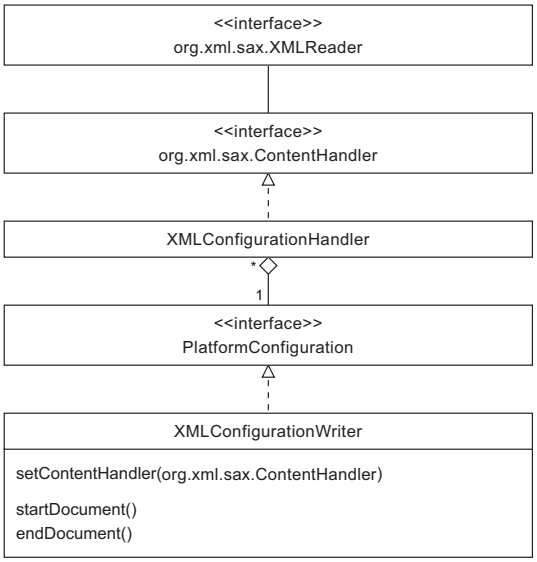

Figure 8.4: UML class diagram of the stream-based configuration

The event-based processing of the input stream relieves the system from storing the configuration in some container. This helps particularly save resources when dealing with large configurations. Following the decorator design pattern (Gamma et al., 1995, p. 175), implementations of the `PlatformConfiguration` interface can be decorated with other classes that process the configuration elements before adding them to the underlying configuration. For instance, process definitions could be filtered or modified.

The `XMLConfigurationWriter` helps persist configuration elements in the XML format. It implements the `PlatformConfiguration` interface (Figure 8.4). This writer does not simply write an XML file. Actually, it does not write any file at all. Instead, the writer simply generates the SAX events for the respective XML elements as if it were an XML parser. A SAX `ContentHandler` can be registered in order to handle the generated events. Therewith, this class is much more versatile than a simple writer.

The registered content handler can be used in order to forward the generated SAX events directly to another XML processor. Hence, it is possible to build XML pipelines without an indirection over the file system. In order to transform the XML events back into configuration elements (e.g., after some XSL transforms), the `XMLConfigurationHandler` can be employed. Such a handler could even be registered directly as a content handler for this writer (although there are very rare occasions, if any, where this seems reasonable in practice).

A `java.xml.transform.sax.TransformerHandler` can be created with the help of the `java.xml.transform.sax.SAXTransformerFactory` to write an XML file. This handler must be registered as the content handler of the `XMLConfigurationWriter`. The destination of the data can be specified with the `java.xml.transform.Result` of the handler.

Before adding configuration elements to the `XMLConfigurationWriter`, `startDocument()` must be called. After adding all configuration elements, `endDocument()` must be called in order to finish the document. If the `endDocument()` method is not called, no valid XML is generated by the writer. Note that this method does not implicitly call some `close()` method of some underlying stream. Operating only on an event interface, this class does not have access to such a method.

References

Bellifemine, F., Caire, G., & Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE*. Chichester, UK: John Wiley & Sons.

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The Unified Modeling Language User Guide* (2nd ed.). Amsterdam, The Netherlands: Addison-Wesley Longman.

Foundation for Intelligent Physical Agents. (2002a). *FIPA ACL Message Representation in String Specification* (Standard No. fipa00070). Geneva, Switzerland.

Foundation for Intelligent Physical Agents. (2002b). *FIPA ACL Message Representation in XML Specification* (Standard No. SC00071E). Geneva, Switzerland.

Foundation for Intelligent Physical Agents. (2002c). *FIPA ACL Message Structure Specification* (Standard No. SC00061G). Geneva, Switzerland.

Foundation for Intelligent Physical Agents. (2002d). *FIPA Communicative Act Library Specification* (Standard No. SC00037J). Geneva, Switzerland.

Foundation for Intelligent Physical Agents. (2002e). *FIPA Contract Net Interaction Protocol Specification* (Standard No. SC00029H). Geneva, Switzerland.

Foundation for Intelligent Physical Agents. (2002f). *FIPA Request Interaction Protocol Specification* (Standard No. SC00026H). Geneva, Switzerland.

Foundation for Intelligent Physical Agents. (2004). *FIPA Agent Management Specification* (Standard No. SC00023K). Geneva, Switzerland.

Freeman, S., & Pryce, N. (2009). *Growing Object-Oriented Software, Guided by Tests*. Amsterdam, The Netherlands: Addison-Wesley Longman.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Amsterdam, The Netherlands: Addison-Wesley Longman.

Huhns, M. N., & Stephens, L. M. (1999). Multiagent Systems and Societies of Agents. In G. Weiss (Ed.), *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence* (pp. 79–120). Cambridge, MA, USA: MIT Press.

Kay, M. (2008). *XSLT 2.0 and XPath 2.0* (4th ed.). Indianapolis, IN, USA: Wiley Publishing.

McLaughlin, B. (2001). *Java & XML* (2nd ed.). Sebastopol, CA, USA: O'Reilly & Associates.

Odell, J., Parunak, H. V. D., & Bauer, B. (2000). Representing Agent Interaction Protocols in UML. In P. Ciancarini & M. Wooldridge (Eds.), *1st International Workshop on Agent-Oriented Software Engineering (AOSE 2000)* (pp. 121–140). Limerick, Ireland: Springer-Verlag.

Schuldt, A. (2011). *Multiagent Coordination Enabling Autonomous Logistics*. Heidelberg, Germany: Springer-Verlag.

Schuldt, A., Gehrke, J. D., & Werner, S. (2008). Designing a Simulation Middleware for FIPA Multiagent Systems. In L. Jain et al. (Eds.), *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT 2008)* (pp. 109–113). Sydney, Australia: IEEE Computer Society Press.

Shore, J. (2004). Fail Fast. *IEEE Software*, *21*(5), 21–25.