Transceiver Framework &
MDP: A Motif Detector and Predictor

Grzegorz Stepien
grzegorz.stepien@rwth-aachen.de

December 2018

Contents

1 Introduction **1**
- 1.1 Challenges of time series stream analysis . . . 2
- 1.2 Motifs as patterns of interest . . . 2
- 1.3 Our contribution . . . 3
- 1.4 Structure of this work . . . 4

2 Transceiver framework **5**
- 2.1 UML naming and color convention . . . 5
- 2.2 Concurrent tasks . . . 6
- 2.3 Transmitters, receivers and transceivers . . . 9
  - 2.3.1 Lifecycles . . . 11
  - 2.3.2 Example . . . 15
- 2.4 Managing multiple parallel tasks with `MultiTask`s . . . 15
  - 2.4.1 Lifecycle . . . 17
  - 2.4.2 Example . . . 18
- 2.5 Managing consecutive `MultiTask`s with `MultiTaskChain`s . . . 18
  - 2.5.1 Lifecycle . . . 22
  - 2.5.2 Examples . . . 24
- 2.6 Special `MultiTask`s . . . 27
  - 2.6.1 Branching data flows . . . 27
  - 2.6.2 Demultiplexer: From single source transmitter to multiple output transmitters . . . 29
  - 2.6.3 Multiplexer: From multiple input receivers to single target receiver . . . 32

3 MDP: Motif detector & predictor **37**
- 3.1 Color convention . . . 37
- 3.2 Data types . . . 37
- 3.3 SDG demultiplexer . . . 38
- 3.4 Motif detector transceiver . . . 39
  - 3.4.1 Matrix profile based motif search . . . 41
- 3.5 Continuous F1 evaluation multiplexer . . . 42
- 3.6 Ordering multiplexer for continuous language model training and evaluation . . . 43
- 3.7 MDP example configuration . . . 45

Bibliography **47**

1 Introduction

Today's rapid progress of data storage and processing technology opens up a wide spectrum of possibilities for gaining new insights into real world phenomena. The prevalence of sensors in devices like smartphones, cars and industry machinery as well as technologies like the Internet lead to an ever increasing amount of data being continuously produced. While entailing great potential for new insights, it is the sheer amount and heterogeneity of data which makes the task of extracting meaningful information far from trivial. Consequently, over the last decades *Data Science* emerged as an interdisciplinary field that aims at applying methods from statistics, signal processing, information theory and computer science in order to tackle various aspects of big data analysis. Areas of application range from genome research and personalized medicine [1], particle detection in collider experiments [2, 3] through to prediction of earthquakes [4], crimes [5], business bankruptcies [6], machinery failures [7] (known as *predictive maintenance*) and many others.

Time series stream analysis is a subfield of Data Science. The term "time series" refers to data which may be represented as a function of discrete points in time. The term "stream" indicates that new data is being continuously produced and needs to be analyzed in an online, real-time fashion (as opposed to, for example, historic time series which are stored as a whole). Typical sources of time series streams are *sensors*. In certain time intervals, the latter produce some real valued measurements (like temperature, pressure, acceleration, ...) or discrete labels (like "defect", "no defect", ...) reflecting real world phenomena. The analysis of time series streams typically serves the purpose of supporting some sort of (semi-) automated decision making and event detection & prediction.

Current efforts to implement the concept of *Industry* 4.0[8] are a very prominent example for the importance of efficient analysis of time series streams. This hoped-for "4th industrial revolution" aims at minimizing production cost while simultaneously maximizing production flexibility in order to enable higher individualization of products at lower costs. At an intra-factory level, this entails the vision of a self organizing *smart factory* that employs interconnected sensors, embedded production systems and RFID chips on each product. Communication between those components is realized by an *Internet of things*-type infrastructure. This allows to create a (semi-) bijective mapping between the physical production environment and a digital representation of the former. Systems with this type of close physical-virtual interrelation are called *cyber-physical systems*. Input from the physical side of said mapping may be continuously fed into data processing methods from fields like pattern recognition, data mining and/or machine learning. The results of the former may then be used by an automated decision making process, the output of which is fed back into the physical side in order to trigger some production reorganization/optimization. At an inter-factory level, on the other hand, a main goal of Industry 4.0 is to digitally interconnect factories and businesses in order to enable automated and more flexible supply-chains necessary for the aforementioned intra-factory flexibility.In theory, such an infrastructure both on intra- as well as inter-factory levels would allow for highly dynamical production processes with a high level of decentralized self organization and little need for human oversight.

1.1 Challenges of time series stream analysis

The need for efficient methods for the analysis of live sensor data streams is apparent, yet comes with a number of difficulties:

1. In a typical multi-sensor scenario we are dealing with multiple parallel data streams where meaningful patterns and their correlations might be asynchronously distributed across several streams.

2. Data streams might be made up of numerical as well as non-numerical (i.e., symbolic) label data streams. Dealing with heterogeneous, interrelated streams is a non-trivial challenge since not every technique is suitable for every data type.

3. In practice we often require the sensor data to be evaluated in an online, real time fashion in order to be able to react as fast as possible to the emergence of relevant time series patterns.

4. We cannot always assume the ideal case that one of the label streams contains some form of event ground truth which we could use to train a predictor for future events. But even in a scenario without ground truth, our system should at least be able to detect and predict meaningful repeating patterns.

5. A more practical, but nonetheless important issue when developing new methods for data analysis is the need for a scalable framework able to concurrently handling and forwarding data from each stream across various processing steps and providing a simple API for:

   - Defining the data processing infrastructure (order of data processing steps, forks and joins of intermediate processed streams, etc.).
   - Plugging in custom methods for each data processing step.

   Ultimately, such a framework should provide the developer with a highly scalable, yet easy to use tool enabling him or her to focus (almost) completely on the implementation of his or her data processing method without having to worry about the technical infrastructure, concurrency issues, etc.

Our *Transceiver Framework* is a JAVA framework which provides exactly those functionalities described in Item 5. Embedded in the transceiver framework, we implemented a *Motif Detector and Predictor (MDP)* which addresses the other issues in the enumeration above.

1.2 Motifs as patterns of interest

The MDP focuses on so called *motifs* as patterns of interest. Given a numeric sequence, a motif is a subsequence which appears at least twice in said sequence, i.e., a subsequence that is *similar* to a different subsequence with respect to some metric (as we will see in Section 3.4, in our case we apply the z-normalized Euclidean distance). The motivation for choosing motifs as patterns of interest is the assumption that motifs do not simply occur at random, but rather that the order of their occurrence holds valuable information for the detection and prediction of certain real world events. Our core assumptions on the nature of our data may therefore be summarized as follows:

1. Some *entity* is the subject of repeated measurements resulting in a number of numerical and/or symbolic time series streams.

2. Real world *events* related to the entity of interest are reflected by the occurrence of:
   - One or more specifically shaped, finite subsequences in the numerical data streams. Such repeating subsequences are called *motifs*.
   - One or more specific labels in the symbolic data streams.
3. Motifs and labels might be distributed among several sensor data streams and might not all occur at the same time. Some of them might occur before and some after the actual event. Earthquakes, for example, are usually foreshadowed by imperceptible preliminary tremors [9]. Similarly, Machine defects might announce themselves in one form or another (increasing vibration, a specific sound, pressure drop/increase, or a specific combination such patterns).
4. Events of the same type or nature produce similar sets of Motifs with similar temporal delays between them.

1.3 Our contribution

Building on the observations from the last two sections, the contributions of this work comprises of two parts:

1. Addressing Item 5 from Section 1.1, we implemented a modular and concurrent JAVA *transceiver framework* for the parallel processing of multiple data sources. It is a conveyor belt like architecture made up of data sources which we call *transmitters*, data sinks called *receivers* and data processors called *transceivers*. To allow for a high degree of scalability, each transmitter, receiver and transceiver runs in its own thread and the framework makes sure that they communicate with each other in a thread safe manner. The framework allows for complex architectures containing data flow forks, joins and multiple subsequent data processing stages. Working with this framework, all that a developer has to do is:
   - Implementing a transmitter interface to the data source (like, for example, a sensor, a file containing historical data or a synthetic data stream generator like our SDG).
   - Implementing a transceiver interface with the actual data processing procedures.
   - Implementing a receiver interface to the data sink (where, for example, the results of the data processing may be joined and serve as the basis for some automated decision making process).
   - Defining the actual architecture of the data flow (i.e., which transmitter shall be connected to which transceiver and which transceiver to which receiver, etc.).

   Furthermore, the framework provides a special driver class that allows to configure and instantiate arbitrary data flow architectures via a simple JSON file.
2. We implemented a data stream analysis tool we refer to as *Motif Detector and Predictor (MDP)*. It combines and extends methods from areas like motif detection and natural language processing in order to address each of the other challenges listed in Section 1.1. MDP itself is embedded in our transceiver framework.

   Figure 1.1 contains a simplified UML 2.0 object diagram depicting its basic control and data flow in case of two numeric and two symbolic input data streams. The latter are provided by an interface to an concurrently running instance of our Synthetic Datastream

Generator (SDG) [10] and each dimension's stream is forwarded to a unique transmitter which feeds its stream into subsequent processing stations. The core idea is to symbolize the incoming numerical data stream which means that we transform each numeric input stream into a symbolic stream. Symbolization is realized by scanning the each numerical stream for repeating subsequences (motifs) and labeling each data point by whether it is part of a motif or not. In the last step we perform an ordered-merge of the resulting symbolic streams resulting in a single stream where elements are ordered in an ascending manner with respect to their timestamps. The latter is fed into a continuously learning probabilistic language model (LM). This model, in turn, allows us to predict the most probable next label(s).

Note that, given we are able to search for motifs in a real time fashion (which, as we will see, is the case), such an architecture is indeed inherently able to cope with each of the challenges listed in Section 1.1 (asynchronicity, heterogeneity, real time processing and pattern detection in the absence of ground truth). In a practical scenario, the numeric streams might be generated by sensors from some machinery and the symbolic streams might contain information about the occurrence of certain defects. By being able to predict labels from the symbolic stream based on those from the symbolized numeric ones, one has the means to predict such defects before they actually occur.

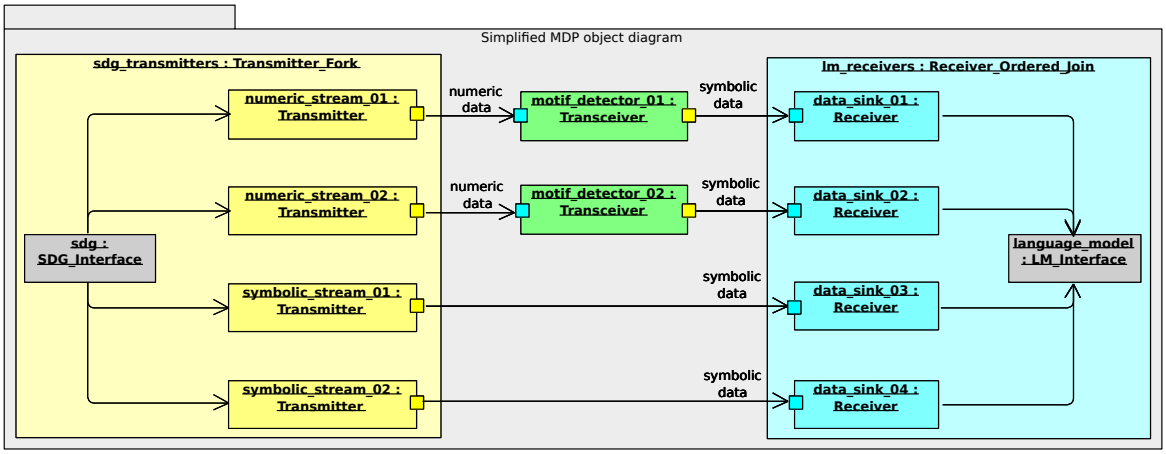

Figure 1.1: UML 2.0 object diagram depicting a greatly simplified example of our MDP as a horizontally structured set of parallel data processing lines.

1.4 Structure of this work

In the following chapters, we describe the transceiver framework's most important classes, their behavior and lifecycles based on UML 2.0 class, state and object diagrams. We then continue to provide an extensive description of our MDP implementation, how its components interact with each other and how they are embedded into the transceiver framework (again, based on UML 2.0 class and object diagrams).

We implemented the Transceiver Framework and the MDP in JAVA. Both are available at [11].

2 Transceiver framework

In this chapter, we present the main components of our *Transceiver Framework*. The latter provides an infrastructure for parallel, multistage processing of data and data streams in particular. We describe each module's inner structure on the basis of 2.0 class diagrams and its lifecycle based on state diagrams. We also provide object diagrams to in order to illustrate each module's application via a concrete example. The UML diagrams are closely guided by the actual code structure. Nonetheless, they do not represent a complete technical documentation. Their main purpose is to facilitate an understanding of how the the framework works by providing a simplified view of our implementation.

The transceiver framework is implemented in JAVA.

2.1 UML naming and color convention

We use a JAVA like terminology throughout this chapter. Table 2.1 contains on overview of our naming convention and our diagrams use the color convention established in Table 2.2.

Notation:	Meaning:
`Abstract*`	Represents abstract classes with at least one method that has to be overridden by subclasses.
`Generic*`	Generic, non-application specific implementations of abstract classes.
`TEMPLATE_PARAMETERS`	Template parameters are always written in upper case. The template parameters introduced in this chapter are the same we use in our actual implementation.
{`Constraint`}	Constraints are expressed in curly braces.
`Type<Template_Type>`	We either set template parameters via the first, JAVA like notation or by using the second, UML binding stereotype notation. We typically use the former to specify the type of member variables and the latter for the specification of a subclass type.
≪`bind T → Template_Type`≫	
Array indices start at 1.	

Table 2.1: Our UML naming convention.

Color:	Refers to:
Red	References to JAVA interfaces.
Yellow	Interfaces, classes and packages related to transmitters.
Blue	Interfaces, classes and packages related to receivers.
Green	Interfaces, classes and packages related to transceivers.
Grey	None of the above

Table 2.2: Our UML color convention.

2.2 Concurrent tasks

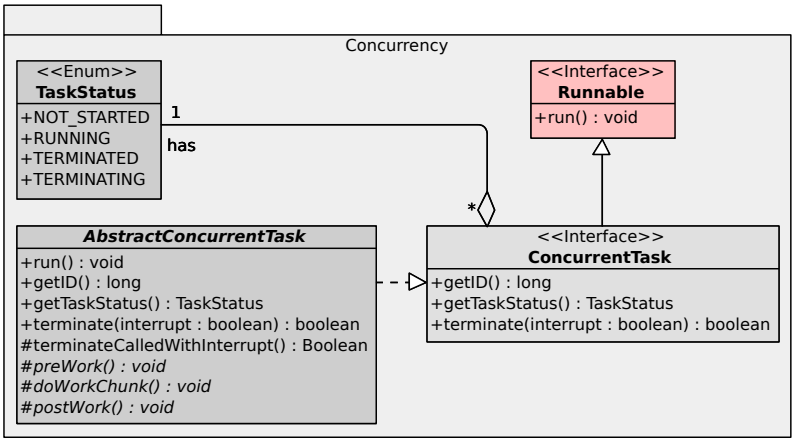

Figure 2.1: UML 2.0 class diagram depicting the concurrency interface and its abstract implementation.

Consider Figure 2.1: The `ConcurrentTask` interface extends the JAVA `Runnable` interface and therefore represents a concurrently executed task[1]. Almost all interfaces introduces in this chapter extend `ConcurrentTask`, i.e. are executed concurrently in their own thread. A `ConcurrentTask` instances have an ID (`getID()`), task status (`getTaskStatus()` and may be terminated via `terminate(...)`. The task status is represented by an `enum TaskStatus` which contains the following fiels:

- `NOT_STARTED`: The `ConcurrentTask` instance has not been submitted to a thread so far.
- `RUNNING`: The `ConcurrentTask` instance's `run()` is being executed by a thread and `terminate(...)` has not been called yet.
- `TERMINATING`: The `ConcurrentTask` instance's `run()` is being executed by a thread and `terminate(...)` has been called.
- `TERMINATED`: The `ConcurrentTask` instance's `run()` has finished after the termination procedure.

`AbstractConcurrentTask` is an abstract implementation of `ConcurrentTask` which takes care of the overall lifecycle of instances of the latter. All implementations of `ConcurrentTask` should extend `AbstractConcurrentTask`. Its general lifecycle is depicted in Figure 2.2:

[1]The JAVA `Runnable` interface provides a single `run()` method which subclasses have to override in order to implement the functionality that shall be executed concurrently.

1. Initialization via `preWork()`: After the `AbstractConcurrentTask` instance has been created and submitted for execution, its corresponding thread enters the `run()` method. The task status switches to `RUNNING` and the abstract method `preWork()` is called. Implementing subclasses override this method in order to realize some application dependent initialization procedures.

2. Iterative `doWorkChunk()` calls: Next, the thread enters a loop that repeatedly calls the abstract `doWorkChunk()` method. Implementing subclasses override this method in order to realize the actual, aplication depended work that the instance shall perform. The loop is executed for as long as the task status is `RUNNING`.

   A concurrent call to `terminate(interrupt)` switches the task status to `TERMINATING`. In addition, if the `interrupt` argument is `true`, the thread executing the instance's `run()` method is interrupted, thus interrupting any potential blocking method calls the latter might currently be in during a `doWorkChunk()` execution.

3. Termination cleanup via `postWork()`: After the executing thread leaves the main loop, it enters the abstract `postWork()` method. Implementing subclasses may override this method in order to realize some application dependent cleanup procedure. Whether `terminate(interrupt)` was called with a `true` or `false`, may be discovered from the return value of `terminateCalledWithInterrupt()` (note that the latter returns a `Boolean` with capital 'B'[2]). Before termination, the latter simply returns `null`. The idea is that an interrupted termination should be considered as a "hard exit" where unprocessed data shall be discarded while an uninterrupted termination is a "soft exit" where no new data shall be accepted but the content of, for example, some internal input buffers may still be processed.

[2] JAVA provides wrapper classes for all primitive types. A primitive `true` is represented by the static final `Boolean.TRUE` field and `false` by `Boolean.FALSE`.

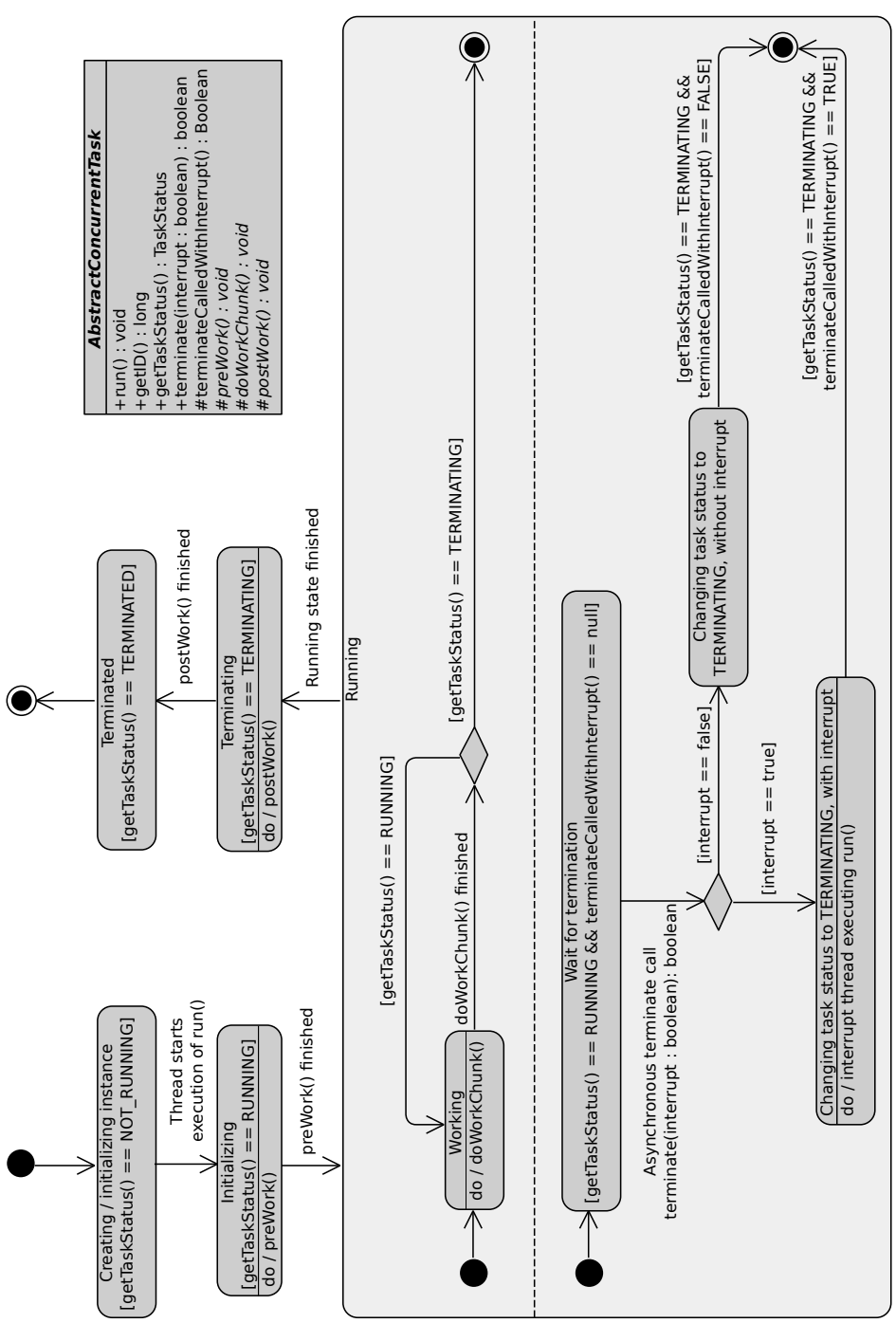

Figure 2.2: UML 2.0 state diagram depicting the lifecycle of `AbstractConcurrentTask` instances.

2.3 Transmitters, receivers and transceivers

Figure 2.3 contains a class diagram depicting the "atomic" processing units in our framework: The `TransmitterTask`, `ReceiverTask` and `TransceiverTask` interfaces each extend `ConcurrentTask` and their respective abstract implementations (depicted on the left) extend `AbstractConcurrentTask`. Each instance of the former therefore has the lifecycle described in the last section. All abstract subclasses implement the `preWork()`, `doWorkChunk()` and `postWork()` methods and themselves introduce new abstract ones.

The interfaces represent the following concepts:

- A `TransmitterTask` instance:
  - Represents an abstract source of data elements of type `DATA_OUT_TYPE`.
  - Is connected to a single (!) `ReceiverTask<? super DATA_OUT_TYPE>` instance via its `setOutConnection(receiverTask)` method. The input data type of the receiver must be of the same or a supertype of `DATA_OUT_TYPE` in order to guarantee type safety.
  - Iteratively retrieves and forwards data to the receiver it is connected to. The data retrieval is implementation dependent.
- A `ReceiverTask` instance:
  - Represents an abstract sink for data elements of type `DATA_IN_TYPE`.
  - Iteratively processes data previously received from a single (!) `TransmitterTask<? extends DATA_IN_TYPE>` instance. The data processing method is implementation dependent.
- A `TransceiverTask` instance: Implements both `TransmitterTask` and `ReceiverTask` and therefore fulfills both roles.

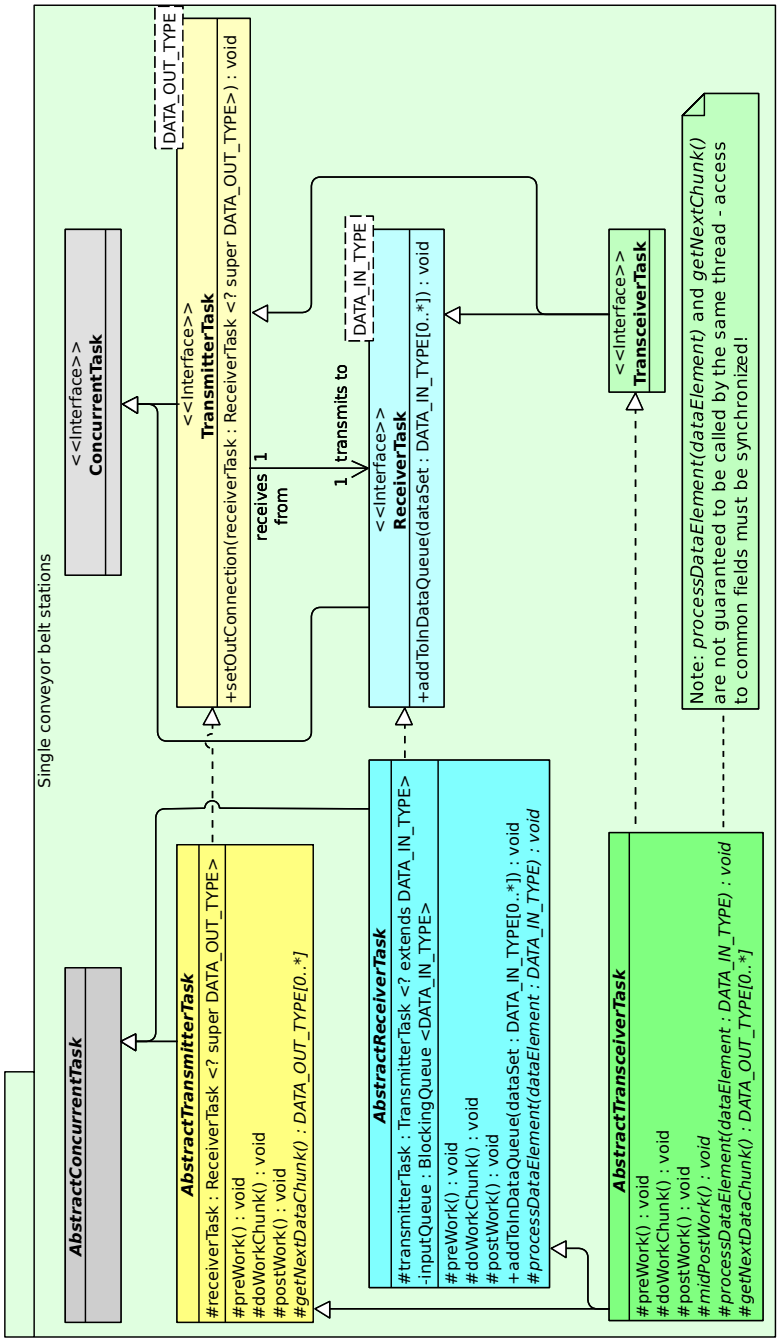

Figure 2.3: UML 2.0 class diagram depicting the transmitter, receiver and transceiver interfaces and their abstract implementations.

2.3.1 Lifecycles

We now continue with a description of the `pre/postWork()` and `doWorkChunk()` implementations of `AbstractTransmitterTask`, `AbstractReceiverTask` and `AbstractTransceiverTask`. See Section 2.2 for how those methods are embedded in the overall lifecycle of those classes.

`AbstractTransmitterTask`

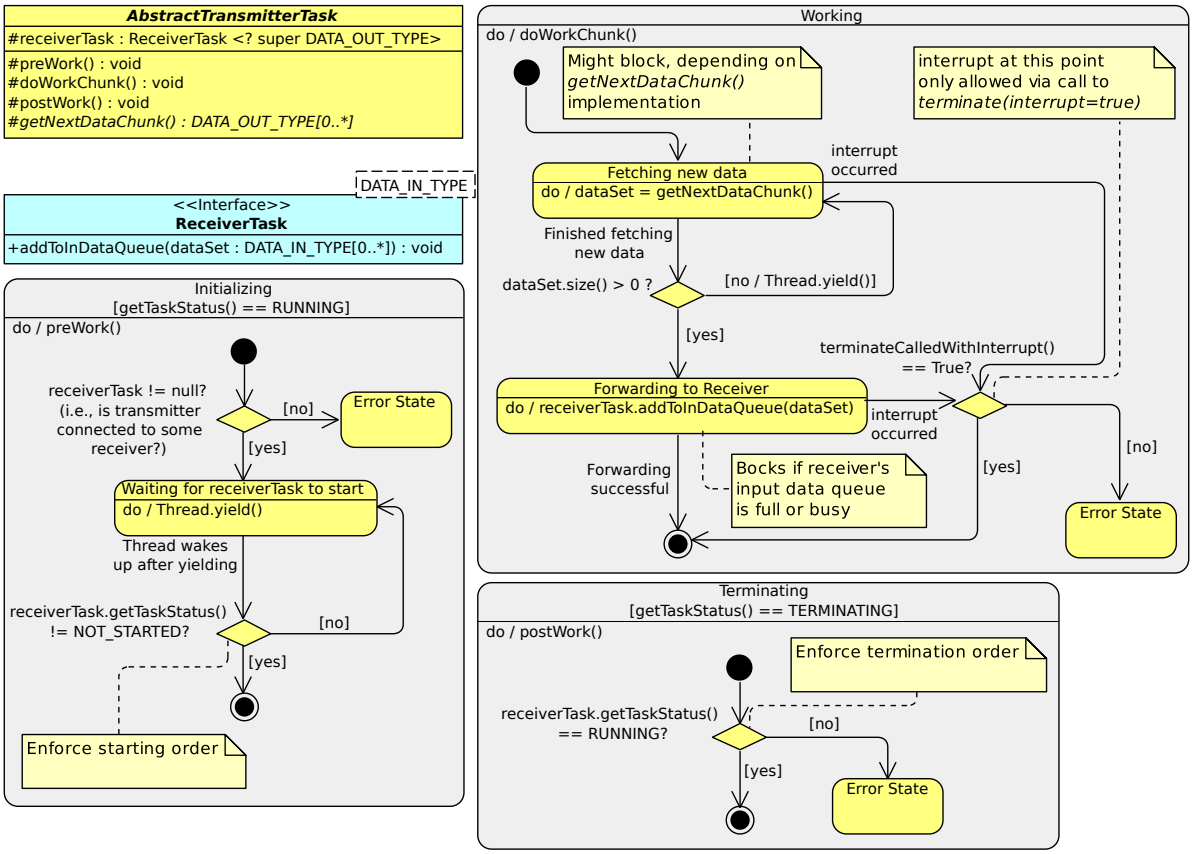

Figure 2.4: UML 2.0 state diagram depicting the lifecycle of `AbstractTransmitterTask` instances.

The `AbstractTransmitterTask` lifecycle is depicted in Figure 2.4:

- `preWork()`: First checks whether the transmitter instance is actually connected to a receiver and produces an error if that is not the case. Note that if the transmitter is connected, an instance of the corresponding receiver is stored in the `receiverTask` field. Next, the method lets its executing thread yield for as long as the target receiver task has not started.
- `doWorkChunk()`: Fetches new data by calling the abstract method `getNextDataChunk()`. Subclasses must override this method in order to implement the actual data fetching and/or generation process. The latter may return an empty data collection or even block. In the former case the executing thread yields followed by a repetition of that process.

  If the data collection is not empty, the data is forwarded to the receiver by a call to the latter's `addToInDataQueue(data)` method. The latter might block if, for example, the

latter's internal data buffer is full. The control flow then leaves the method until its next call.

Note that at this level, uncaught interrupt exceptions are only allowed to be caused by the termination process described in Section 2.2.

- `postWork()`: `AbstractTransmitterTask` instances enforce a certain termination in order to prevent data loss: Transmitters must always be terminated *before* their corresponding receivers.

`AbstractReceiverTask`

The `AbstractReceiverTask` lifecycle is depicted in Figure 2.5:

- `preWork()`: First checks whether some transmitter instance is actually connected to this receiver and produces an error if that is not the case. Note that if the receiver has an in-connection, an instance of the corresponding transmitter is stored in the `transmitterTask` field. Next, the method ensures that there are no cyclic predecessor connections and produces an error if that is the case. It does so by checking its predecessor chain for duplicate entries. A predecessor chain is made up of the current receiver instance, its direct predecessor (i.e., the transmitter connected to it) and, if the latter is also a transceiver, of the latter's predecessors. Note that in order to avoid redundant checks, this check for cycles is only executed by receivers that are not also transceivers (i.e., are guaranteed to be the last element in every predecessor chain).
- `doWorkChunk()`: Fetches a new data element to process from its `inputQueue` field[3] field. The latter is filled by the transmitter via `addToInDataQueue(...)`. Forwards the data element to the abstract `processDataElement(dataElement)` method. Subclasses must override this method in order to implement the actual data processing method. The control flow then leaves the method until its next call.

  Note that at this level, uncaught interrupt exceptions are only allowed to be caused by the termination process described in Section 2.2.
- `postWork()`: Similar to `AbstractTransmitterTask` instances, `AbstractReceiverTask` instances enforce same termination order to prevent data loss: Transmitters must always be terminated *before* their corresponding receivers.

  Given the termination order is correct, the method checks whether `terminate(interrupt)` was called with `interrupt == TRUE` or not. If that is the case, then all remaining data in `inputQueue` is discarded and the constrol flow leaves the method. If not, the method first processes all of the remaining data in the queue. Note that since at this point we ensured that the input transmitter is terminated, the input queue is guaranteed to not receiver any new data. If `processDataElement(dataElement)` is implemented to return after a finite amount of time, this ensures that the processing of the remaining data finishes in a finite amount of time.

  Note that no uncaught interrupt exceptions are allowed to occur during the processing of the remaining data.

[3]The queue is of type `BlockingQueue` which is a JAVA interface for thread-safe queues

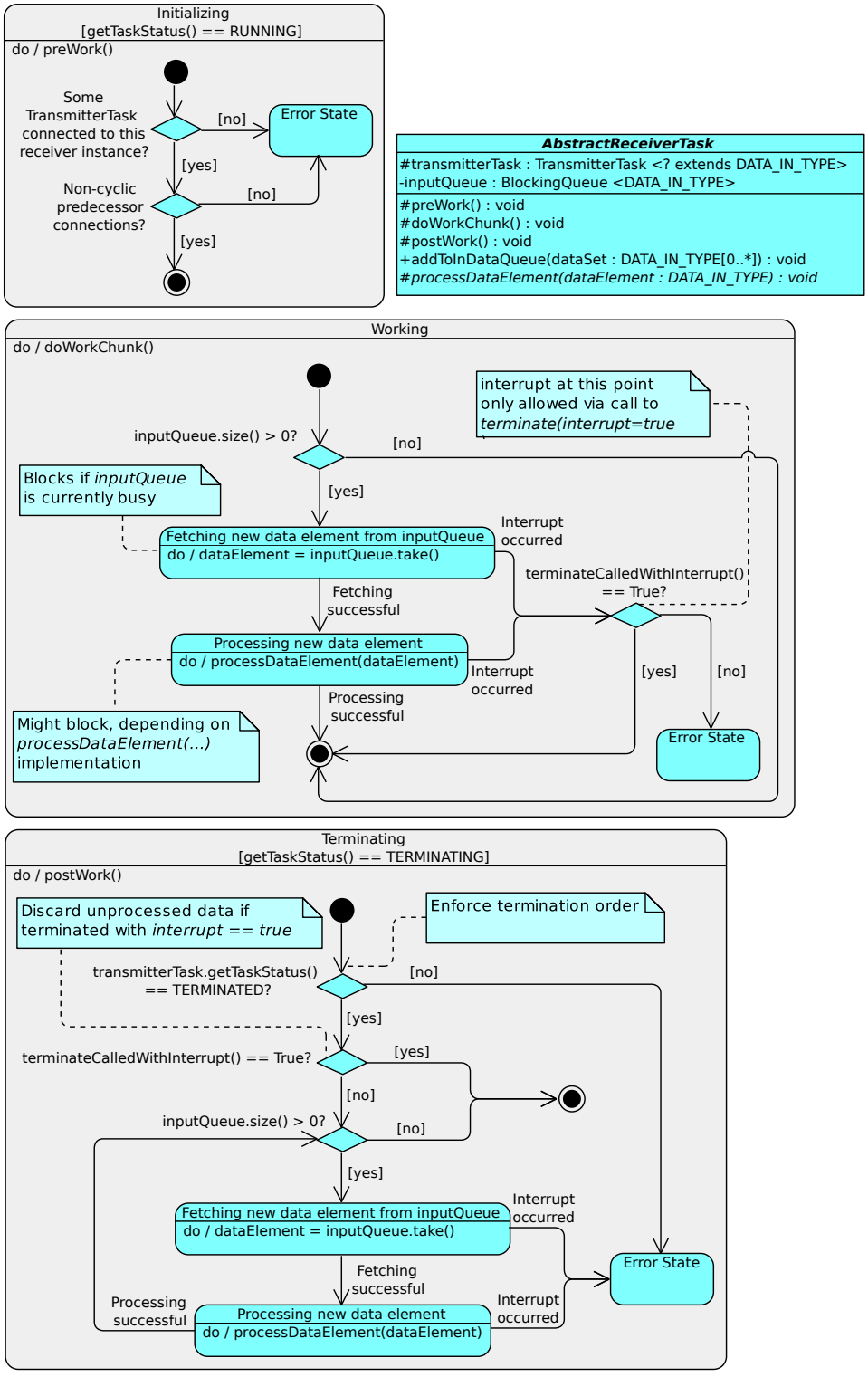

Figure 2.5: UML 2.0 state diagram depicting the lifecycle of `AbstractReceiverTask` instances.

`AbstractTransceiverTask`

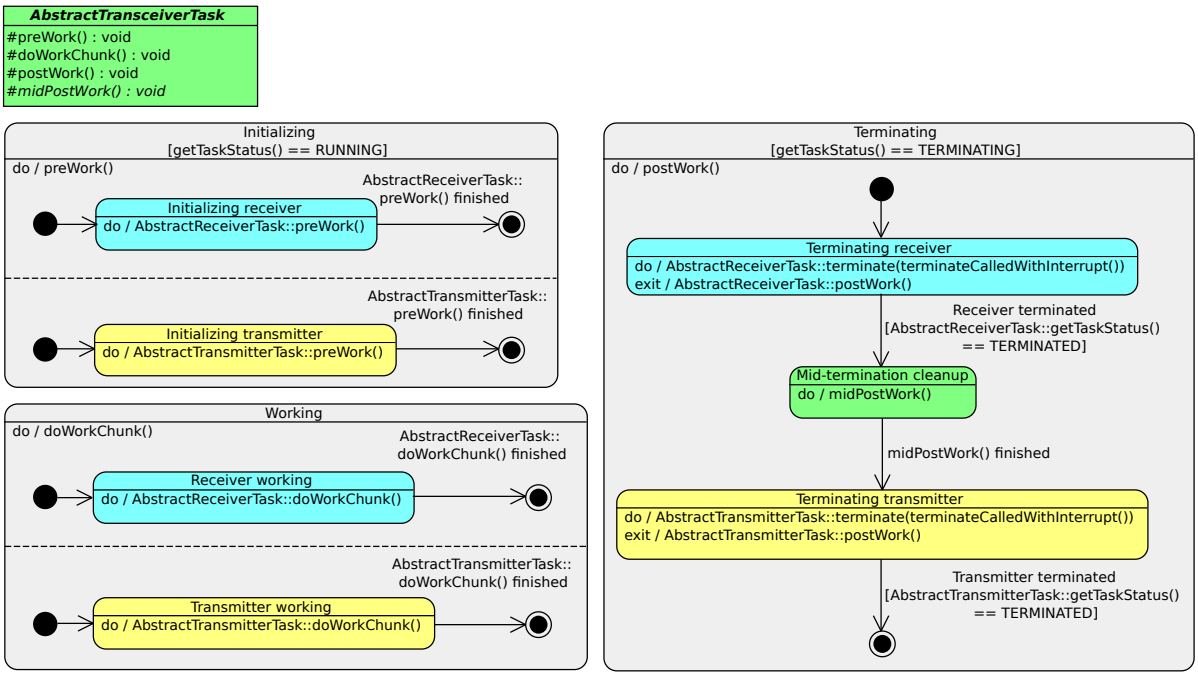

Figure 2.6: UML 2.0 state diagram depicting the lifecycle of `AbstractTransceiverTask` instances.

While JAVA does not support multiple inheritance, `AbstractTransceiverTask` instances are nonetheless implemented in a way to behave in a manner equivalent to inheriting from both `AbstractTransmitterTask` and `AbstractReciverTask` (as depicted in Figure 2.3).

In fact, both aspects are executed in a concurrent manner. This is implemented by each `AbstractTransceiverTask` instance managing a concurrently executed internal instance of `AbstractTransmitterTask` and `AbstractReciverTask` and appropriately forwarding method calls. The execution of `AbstractTransceiverTask` instances therefore actually entails three threads: The first one for the internal `AbstractTransmitterTask` instance, the second one for the internal `AbstractReceiverTask` instance and the third one for the encapsulating `AbstractTransceiverTask` instance itself. Note that the latter is inactive (i.e., sleeping) most of the time in order to avoid waste of CPU time.

Note that this also means that implementations of the transmitter's `getNextDataChunk()` and the receiver's `processDataElement(...)` are executed by two different threads. If both method access a shared field, the access to that field must be appropriately synchronized (future versions of the transceiver framework might provide a safer solution for this potential trap).

This is depicted in Figure 2.6:

- `preWork()`: Concurrently executes the abstract `preWork()` methods of both its superclasses. Both must be implemented by subclasses of `AbstractTransceiverTask`.
- `doWorkChunk()`: Analogous to description of `preWork()`.
- `postWork()`: In order to prevent data loss, the cleanup procedure is *not* executed concurrently as the other two methods. First, the abstract `postWork()` method of the `AbstractReceiverTask` superclass is called. This ensures that no new data reaches the transceiver once that method returns. The thread executing `AbstractTransceiverTask`

then enters the abstract `midPostWork()` method. Implementing subclasses may override this method in order to implement some cleanup of application dependent fields (like, for example, forwarding some remaining data to the receiver to which the transceiver is connected). Lastly, the `postWork()` method of the `AbstractTransmitterTask` superclass is called, thereby shutting the latter down.

2.3.2 Example

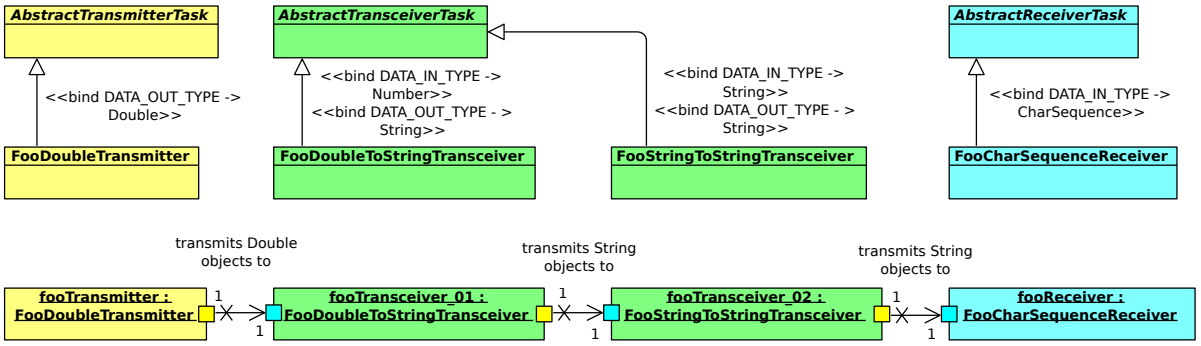

Figure 2.7: UML 2.0 object diagram depicting an example transmitter, receiver and transceiver configuration.

Figure 2.7 depicts an example configuration: Some `FooDoubleTransmitter` generates and transmits some `Double` values to a `FooDoubleToStringTransceiver` instance. In it's receiver role, the latter processes those doubles and generates some `String` values based on them. In its transmitter role, the transceiver fetches those strings from some internal, synchronized data structure and forwards it to an instance of `FooStringToStringTransceiver`. The latter then processes them further (e.g., changes all characters to lowercase) and forwards the result to a `FooCharSequenceReceiver` which writes the values it receives to a file on the hard drive. Note that the latter class expects its input data to be of the same or a subtype of `CharSequence`. In JAVA, this is true for the `String` class which extends `CharSequence`.

2.4 Managing multiple parallel tasks with `MultiTasks`

Manually specifying and connecting parallel transmitter, transceiver and receiver instances involves a certain amount of micro management. To avoid this, `MultiTask` subclasses provide the means to stack several transmitters, receivers or transceivers of the same type in a horizontal fashion and to connect multiple instances of the latter at once via a single method call. Figure 2.8 contains the following instances and classes:

- `MultiTask`: Common superinterface of `MultiTransmitterTask`, `MultiReceiverTask` and `MultiTransceiverTask`. Its `TASK_TYPE` template parameter specifies the type of tasks it manages.
- `AbstractMultiTask`: Abstract implementation of `MultiTask` and common superclass for `GenericMultiTransmitter/Receiver/TransceiverTask`. Manages `<NUM_TASKS>` instances of of `ConcurrentTask` which are stored in the `tasks` field. Implements the common lifecycle of all its subclasses.
- `MultiTransmitterTask`: Restricts `TASK_TYPE` to instances of `TransmitterTask` which themselves must transmit data with a common superclass `COMMON_OUT_TYPE`. This ensures

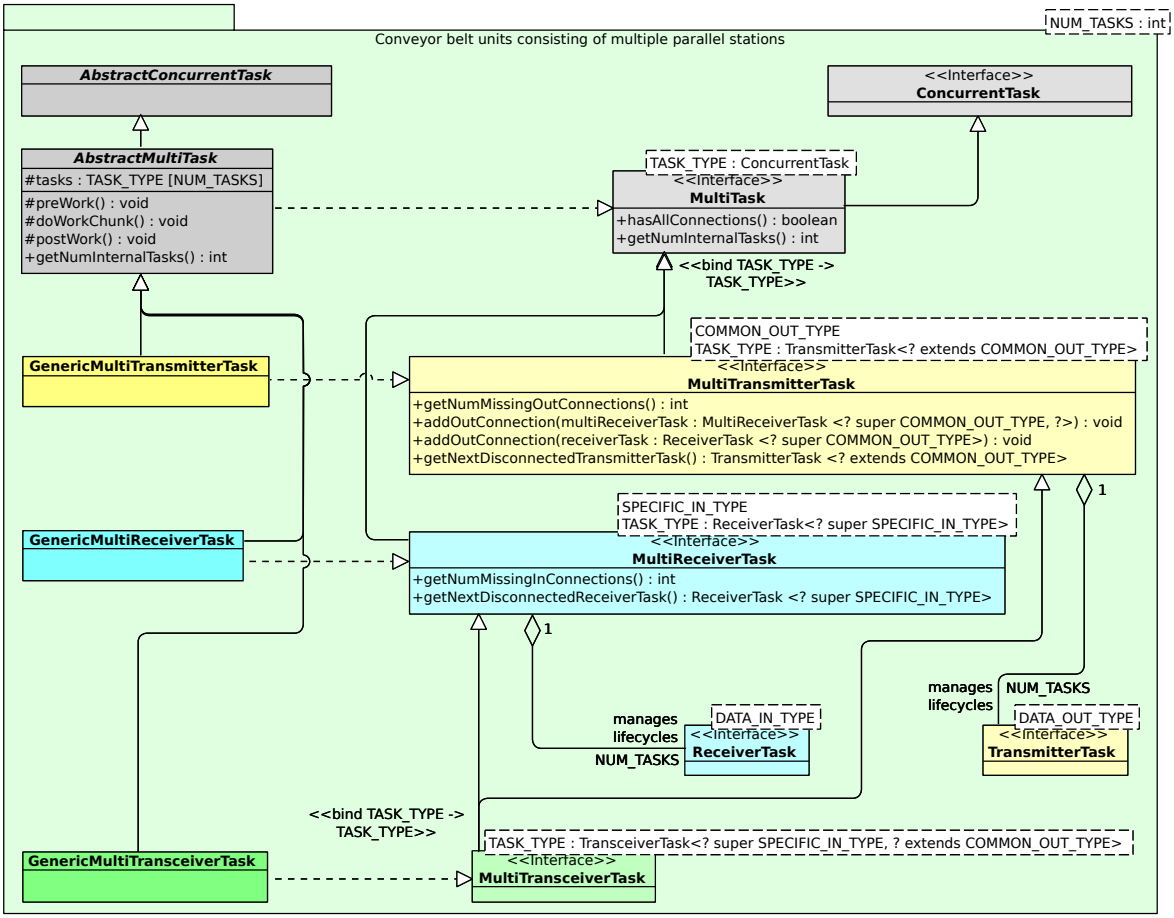

Figure 2.8: UML 2.0 class diagram depicting interfaces and classes for the horizontal stacking of transmitter, receiver and transceiver instances.

that multi transmitters provide a unified interface, including a guarantee that each of their internal transmitters produce data of the same or a subtype of `COMMON_OUT_TYPE`.

- `GenericMultiTransmitterTask`: An implementation of `MultiTransmitterTask`. Implements the following methods:
  * `getNumMissingOutConnections()`: Returns how many of the internal transmitters are not connected to a receiver.
  * `getNextDisconnectedTransmitterTask()`: Returns the next disconnected transmitter from `tasks` (or `null`, if all transmitters are connected).
  * `addOutConnection(receiverTask)`: Connect the next disconnected internal transmitter task to the provided receiver (or produce an error, if all internal connections already set)-
  * `addOutConnecction(multiReceiverTask)`:
    1. $n := \min\{$`getNumMissingOutConnections()`, `multiReceiverTask.getNumMissingInConnections`$\}$;
    2. `For(` $i = 1, \ldots, n$ `):` Connect the next disconnected transmitter task to the

next disconnected receiver task in `multiReceiverTask`;

- `MultiReceiverTask` and `GenericMultiReceiverTask`: Restrict `TASK_TYPE` to instances of `ReceiverTask` which receive data of type `SPECIFIC_IN_TYPE` or a supertype of the latter. This ensures that multi receivers provide a unified interface that guarantees that each of the receivers accepts data of the same type or a subtype of`SPECIFIC_IN_TYPE`. The two depicted `MultiReceiverTask` methods are defined analogous to their transmitter counterparts.
- `MultiTransceiverTask` and `GenericMultiTransceiverTask`: Restrict `TASK_TYPE` to instances of `TransceiverTask`. Since the `MultiTransceiverTask` interface extends both `MultiTransmitterTask` and `MultiReceiverTask`, the all the upper descriptions also apply here.

2.4.1 Lifecycle

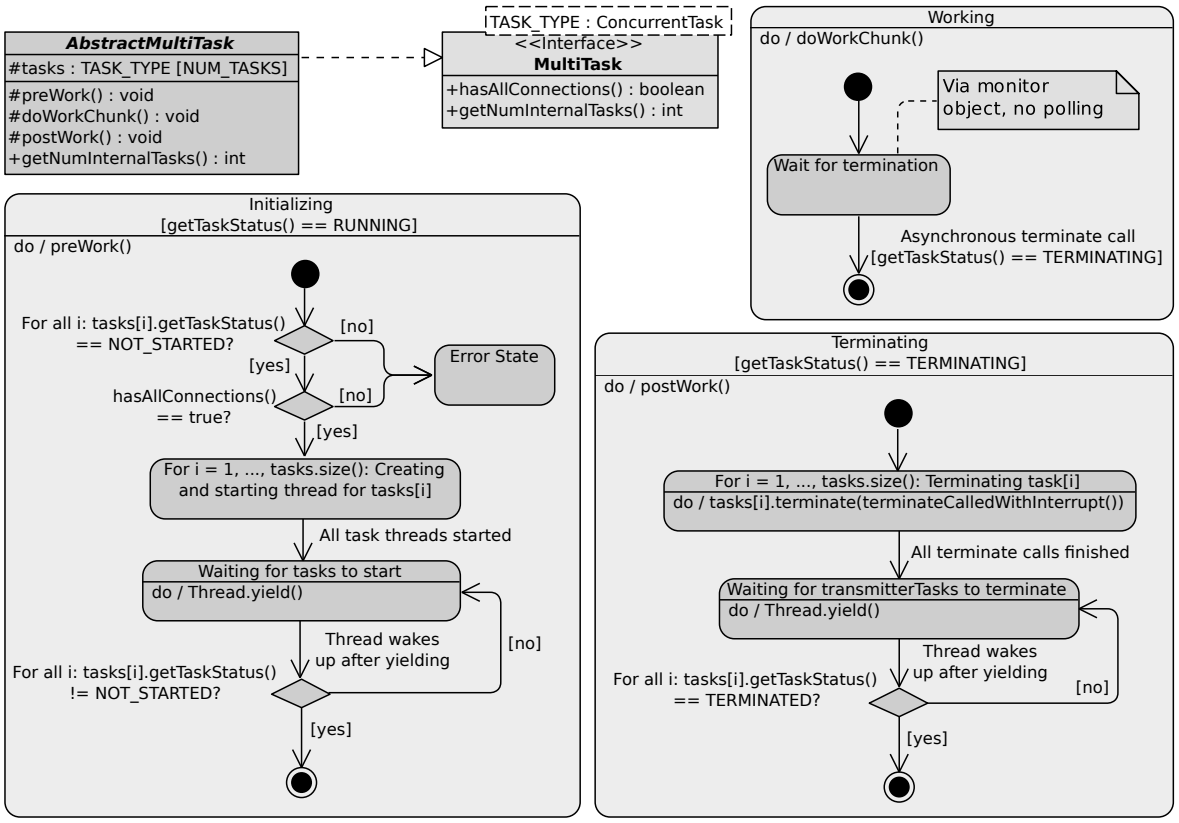

Figure 2.9: UML 2.0 state diagram of depicting the lifecycle of `AbstractMultiTask` instances.

The lifecycle of all subclasses of `AbstractMultiTask` is managed by the latter. It is depicted in Figure 2.9:

- `preWork()`: The lifecycles of all internal tasks are managed by the `AbstractMultiTask` instance. Therefore, all of them must be in the `NOT_STARTED` state at the beginning of `preWork()`. Also, all the internal task's connections must be set (remember that an `AbstractTransmitterTask` and `AbstractReceiverTask` instance produce an error if they are not connected on startup). If one of those two conditions is not fulfilled, an error

is produced. Otherwise, the `AbstractMultiTask` instance creates and starts a thread for each of its tasks and yields until all tasks have left the `NOT_STARTED` state. The control flow then leaves the `preWork()` method.

- `doWorkChunk()`: The thread executing the `AbstractMultiTask` instance sleeps until its `terminate(...)` method is called.
- `postWork()`: Simply calls `terminate(interrupt)` on all internal tasks. The the same `interrupt` argument is used as the one provided to this `AbstractMultiTask` instance's `terminate(...)` method. The executing thread then yields until all internal tasks have entered the `TERMINATED` state.

2.4.2 Example

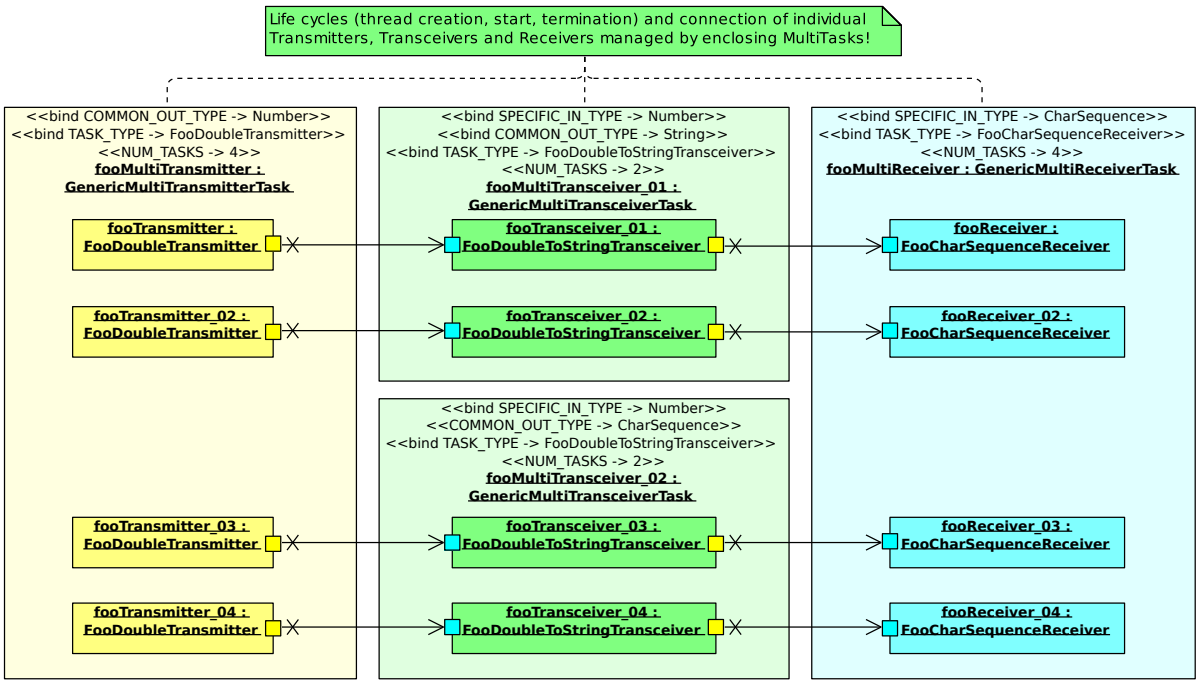

Figure 2.10: UML 2.0 object diagram depicting an example multi transmitter, receiver and transceiver configuration.

Figure 2.10 depicts an example configuration with four `MultiTasks` encapsulating instances of the transmitter, receiver and transceiver classes we define in the last example in Section 2.3.2.

2.5 Managing consecutive `MultiTasks` with `MultiTaskChains`

Now that we have a way of organizing multiple `ConcurrentTask` instances via `MultiTask` instances, we still need to manually connect multiple instances of the latter in order to achieve a certain functionality. However, it may often be the case that multiple consecutive `MultiTasks` solve a particular, well defined task. It would be useful to have some class that is able to encapsulate multiple interconnected `MultiTask` instances into one single instance of the latter. This is what the `MutiTaskChain` interface and its implementations are for. Consider Figure 2.11:

- `MultiTaskChain`: This is the common superinterface of `MultiTransmitterTaskChain`, `MultiReceiverTaskChain` and `MultiTransceiverTaskChain`. `NUM_MTASKS` represents the number of `MultiTask` instances a multi task chain manages. Manages `MultiTask`s whose internal connections are restricted to correspond to a *directed acyclic graph* (DAG) with exactly *one root* and *one leaf*[4]. Two multi task instances are considered to be connected if at least one pair of their internal `ConcurrentTask` instances is connected to each other. The direction of the connection corresponds to the direction of communication (i.e., always from a `TransmitterTask` to a `ReceiverTask` instance). This DAG-constraint ensures that there are no cyclic connections between `MultiTask`s and that we have a single root and a single leaf multi task that may serve as the interface to the `MultiTaskChain`.

  The `ROOT_TYPE` parameter represents the type of the root multi task and `LEAF_TYPE` the type of the leaf multi task.

- `GenericMultiTaskChain`: A generic implementation of `MultiTaskChain` and common superclass for `GenericMultiTransmitter/Receiver/TransceiverTaskChain`. Manages `<NUM_MTASKS>` instances of of `MultiTask` which are stored in the `multiTasks` field. Implements the common lifecycle of all its subclasses.

- `MultiTransmitterTaskChain`: Restricts the `LEAF_TYPE` to `MultiTransmitterTask` subtypes which themselves manage `TransmitterTask` instances that transmit data with a common superclass `COMMON_OUT_TYPE`. Note that `MultiTransmitterTaskChain` also extends `MultiTransmitterTask<? extends COMMON_OUT_TYPE>` and therefore may also act in that role. In fact, calls to the chain's inherited multi transmitter task methods are forwarded to the chain's leaf.

- `GenericMultiTransmitterTaskChain`: A subclass of `GenericMultiTaskChain` which also implements `MultiTransmitterTaskChain`. Its sole purpose is to impose the latter's type restrictions on the former.

- `MultiTransmitterTaskChain`: Restricts the `ROOT_TYPE` to `MultiReceiverTask` subtypes which themselves manage `ReceiverTask` instances that receive data with a common subclass `SPECIFIC_IN_DATA`. Note that the `MultiReceiverTaskChain` class also extends `MultiReceiverTask<? super SPECIFIC_IN_TYPE>` and therefore may also act in that role. This is realized by forwarding calls to the chain's inherited multi receiver task methods to the chain's root.

- `GenericMultiReceiverTaskChain`: A subclass of `GenericMultiTaskChain` which also implements `MultiReceiverTaskChain`. Its sole purpose is to impose the latter's type restrictions on the former.

- `MultiTransceiverTaskChain` and `GenericMultiTransceiverTaskChain`: All the above points also apply here. In order to implement the `MultiTransceiverTask` interface, `MultiTransceiverTaskChain`s impose the additional constraint that the number of receivers in the root multi task must be the same as the number of transmitters in the leaf multi task. The n-th internal receiver from the root multi task and the n-th internal transmitter from the leaf multi task represent the n-th internal transceiver when the chain is used in its multi transceiver role.

- `ClosedMultiTaskChain`: Restricts `ROOT_TYPE` to subtypes of `MultiTransmitterTask` and `LEAF_TYPE` to subtypes of `MultiReceiverTask` thus realizing a chain without any outside connections.

[4]Future versions of the transceiver framework might relax the constraint on root and leaf numbers

- `GenericClosedMultiTaskChain`: A subclass of `GenericMultiTaskChain` which also implements `ClosedMultiTaskChain`. Its sole purpose is to impose the latter's type restrictions on the former.

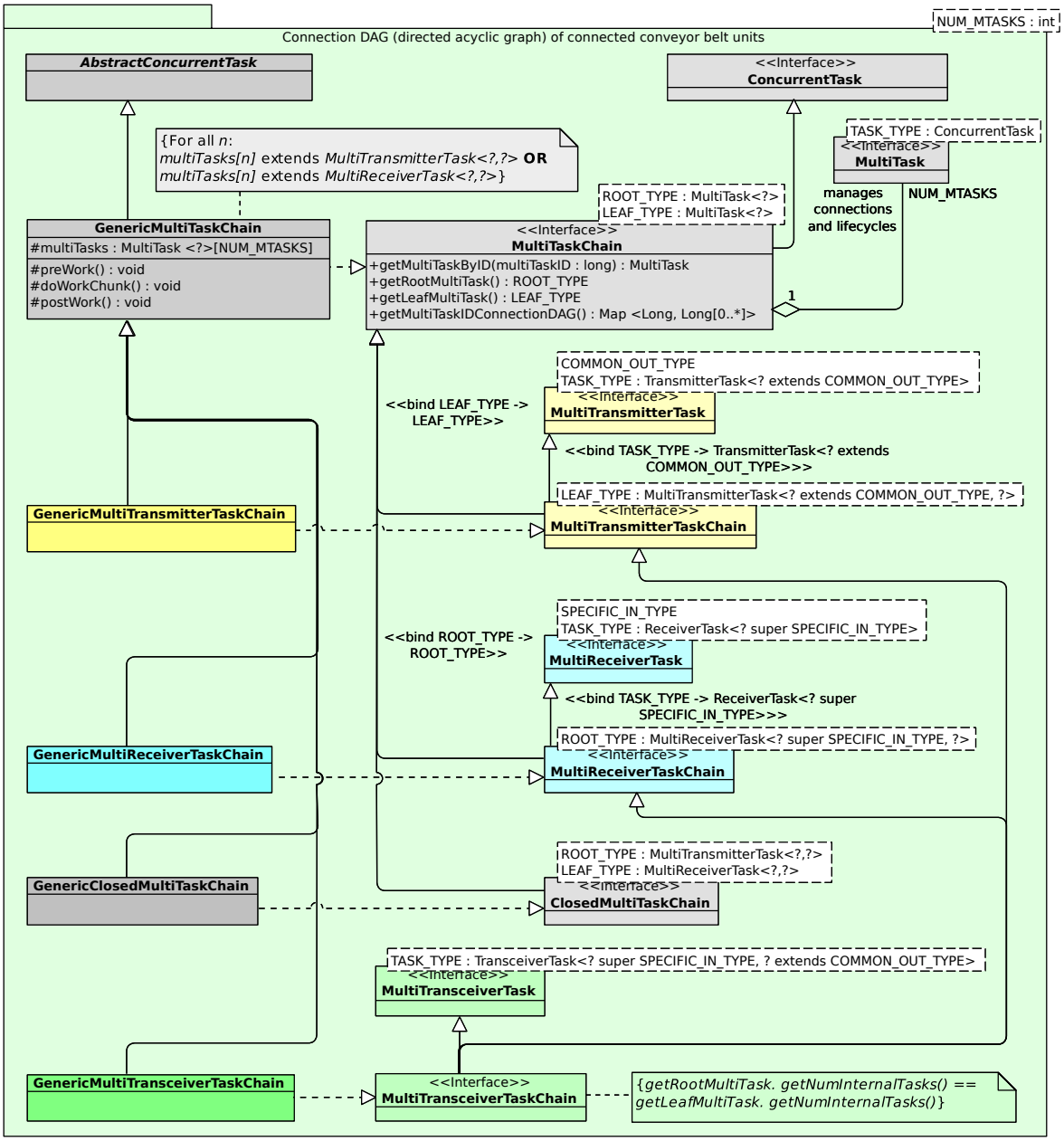

Figure 2.11: UML 2.0 class diagram depicting interfaces and classes for the encapsulation of multiple connected multi transmitter, receiver and transceiver instances.

2.5.1 Lifecycle

The lifecycle of all subclasses of `GenericMultiTaskChain` is managed by the latter. It is depicted in Figure 2.12:

- Before execution: The `GenericMultiTaskChain` constructor receives a set of `MultiTasks` and a `Map<Long, List<Long>>` instance. The first set represents the multi tasks this chain shall manage and the latter represents an map representing the connections between the managed multi tasks. In the map, each multi task is represented by its ID and a mapping $i \to (i_1, \ldots, i_n)$ means that the multi task of ID i shall be connected to the multi tasks of ID i_1 up to i_n. Said map must fulfill certain consistency criteria:
  1. The multi task of ID i must be of type `MultiTransmitterTask` and each of the multi tasks corresponding to IDs $(i_1, \ldots, i_n)$ must be of type `MultiTransceiverTask`. Furthermore, the `COMMON_OUT_TYPE` parameter of the former and the `SPECIFIC_IN_TYPE` of the latter must be compatible (i.e. the former must be the same or a subtype of the latter).
  2. The map must represent a directed acyclic graph with exactly one root and one leaf.

  Note that the above implies that nodes in that graph which are neither leaf nor root must be of type `MultiTransceiverTask`.

  Once DAG-consistency has been established, the chain makes sure that none of the multi tasks has any internal connection set yet as connections are handled by the multi task chain. As a last step, the `GenericMultiTaskChain` connects all its internal `MultiTasks` in an order corresponding to the following deterministic DAG passage:
  * Start at the root.
  * Traverse DAG via depth-first search.
  * If choice of next child node to traverse to is ambiguous, choose in the order they appear in the child list (remember that the provided DAG is represented as a `Map<Long, List<Long>>`).
- `preWork()`: In order to start each internal multi task in an order that ensures that transmitters are always started before the receivers to which they are connected, the multi task chain starts its internal multi tasks in the following order:
  * Create a topological sorting with respect to the DAG. Use the child list order to dissolve any ambiguities in order to create said sorting in a deterministic manner.
  * Iterate through the internal multi tasks in the order of said sorting, submit each one to a thread and wait for it to leave the `NOT_STARTED` state before moving to the next one.
- `doWorkChunk()`: The thread executing the `GenericMultiTaskChain` instance sleeps until its `terminate(...)` method is called.
- `postWork()`: Simply calls `terminate(interrupt)` on all internal multi tasks. It does so in the same order as the multi tasks were started and waits each time for the current multi task to enter the `TERMINATED` state before moving to the next one. This ensures that transmitter-receiver pairs are terminated in the correct order (see Section 2.3.1). The `terminate(interrupt)` method of each multi task is called with the value returned by the multi task chain's `terminateCalledWithInterrupt()` method (see Section 2.2).

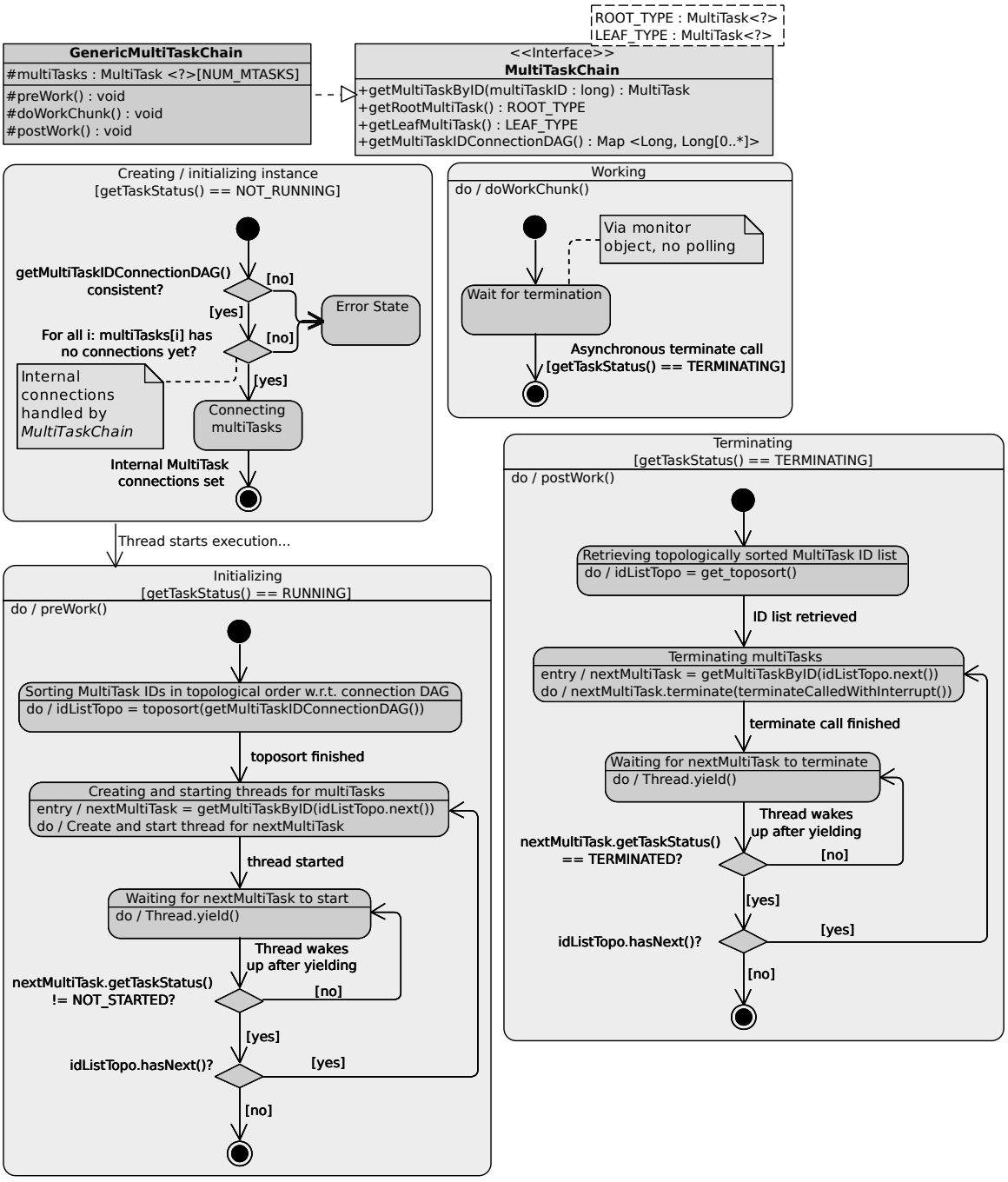

Figure 2.12: UML 2.0 state diagram depicting the lifecycle of `GenericMultiTaskChain` instances.

2.5.2 Examples

Figure 2.13 and depicts an example of the modularization capabilities of `MultiTaskChains`. The innermost chain instance `fooMultiTransceiverChain` is a transceiver multi task chain that encapsulates two connected multi transceiver instances. In its role as a `MultiTransceiverTask` instance, the `fooMultiTransceiverChain` instance is also connected to a multi receiver instance `fooMultiReceiver`. Both are themselves contained within a `MultiReceiverTask` instance `fooMultiReceiverChain` (which is allowed since `fooMultiTransceiverChain` is also of type `MultiTransceiverTask`). The `fooMultiReceiverChain`, on the other hand, is connected to a `MultiTransmitterTask` instance `fooMultiTransmitter`. Lastly, the latter two are again contained within a `ClosedMultiTaskChain` instance.

Figure 2.14 contains a similar example that makes use of a `MultiTransmitterTaskChain`.

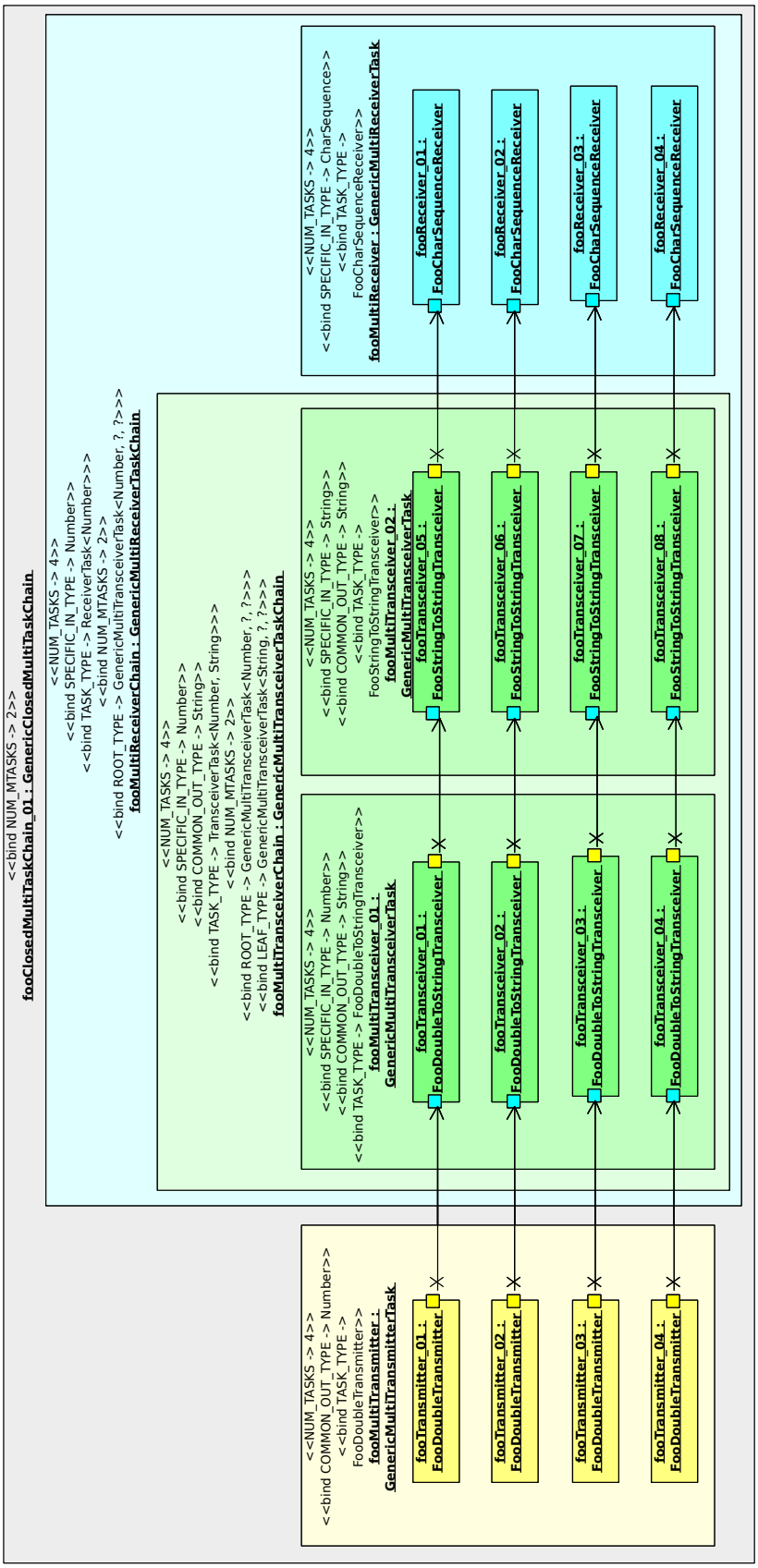

Figure 2.13: UML 2.0 object diagram depicting an example configuration of `MultiTaskChain` instances.

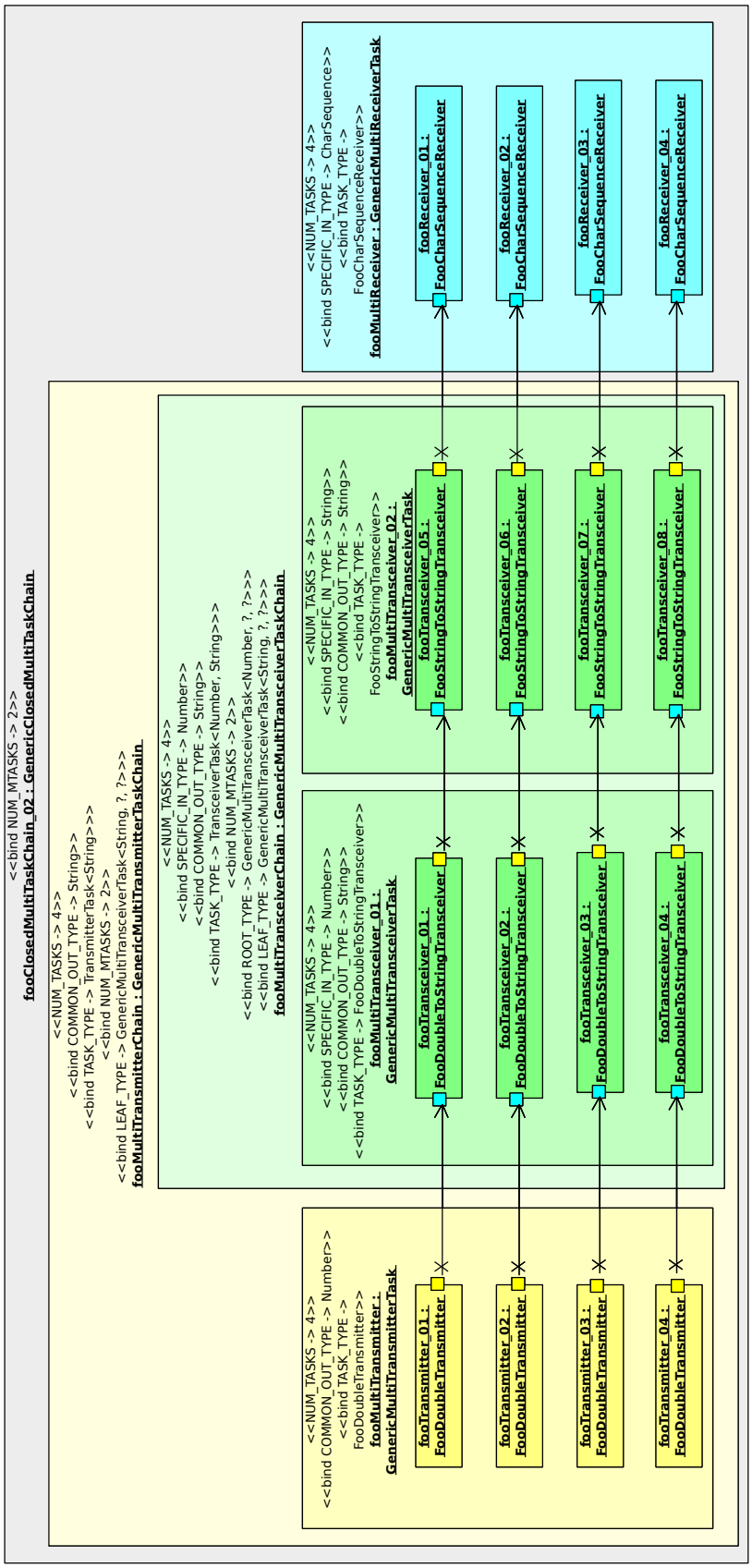

Figure 2.14: UML 2.0 object diagram depicting another example configuration of `MultiTaskChain` instances.

2.6 Special `MultiTasks`

The transceiver framework also contains a number of special `MultiTask` implementations that realize data flow branches, multiplexer and demultiplexer.

2.6.1 Branching data flows

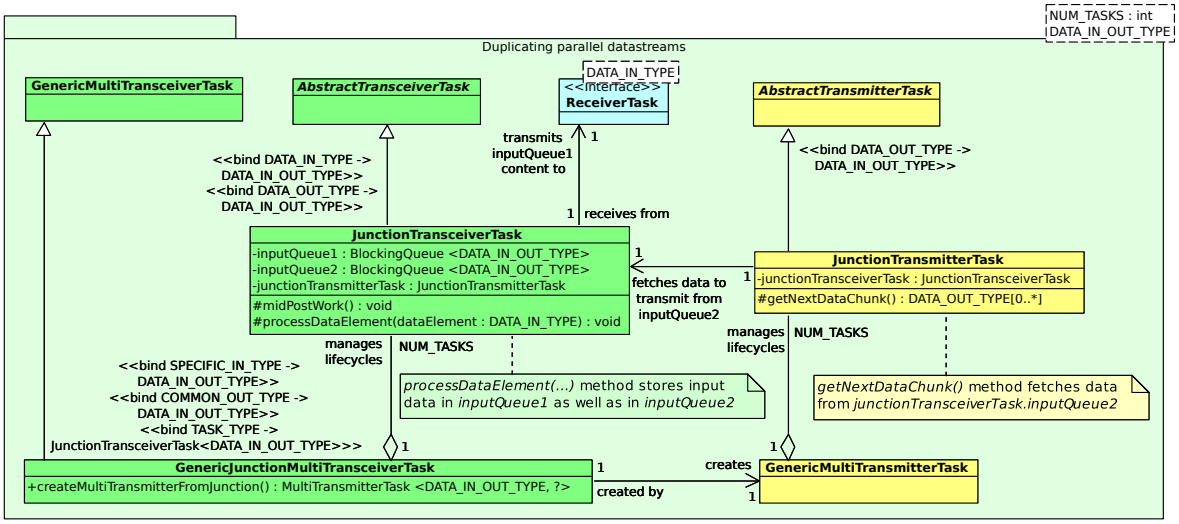

Figure 2.15: UML 2.0 class diagram depicting classes related to the creation of data flow branches.

The `GenericJunctionMultiTransceiverTask` is a special `GenericMultiTransceiverTask` subclass with the purpose of duplicating the incoming data streams (more precisely: by copying their reference, no cloning). Its instances do so by fulfilling two roles: In its role as a `MultiTransceiverTask`, it simply forwards its incoming data to the targets it is connected to. But it also provides the means to create a separate `MultiTransmitterTask` fetches copies of the incoming data in transmits them to its own targets. Figure 2.15 depicts the internal and dependent classes of `GenericMultiTransceiverTask` instances:

- `GenericJunctionMultiTransceiverTask` 1/2: The main junction class. In its role as a `MultiTransceiverTask`, it manages `<NUM_TASKS>` `JunctionTransceiverTask` instances.
- `JunctionTransceiverTask`: As `TransmitterTasks`, those instances have a regular in-connection from some `TransmitterTask` and an out-connection to some `ReceiverTask` - just as described in Section 2.3. Their two abstract methods are implemented as follows:
  - `processDataElement(dataElement)`: Insert a reference to `dataElement` into the `BlockingQueue` `inputQueue1` and `inputQueue2`.
  - `getNextDataChunk()`: Return the next element from `inputQueue1`, if the latter is not empty. Otherwise, return `null`.
- `GenericJunctionMultiTransceiverTask` 2/2:
  Provides the `createMultiTransmitterFromJunction()` method which:
  - May only be called once (although this might be relaxed in future versions).
  - Returns a `GenericMultiTransmitterTask` instance that manages `<NUM_TASKS>` instances of `JunctionTransmitterTasks`.

* The n-th `JunctionTransmitterTask` instance has access to the `inputQueue2` of the n-th `JunctionTransceiverTask` and its `getNextDataChunk()` method fetches its data from this queue.

The returned `GenericMultiTransmitterTask` therefore effectively serves as a data stream junction.

Lifecycle

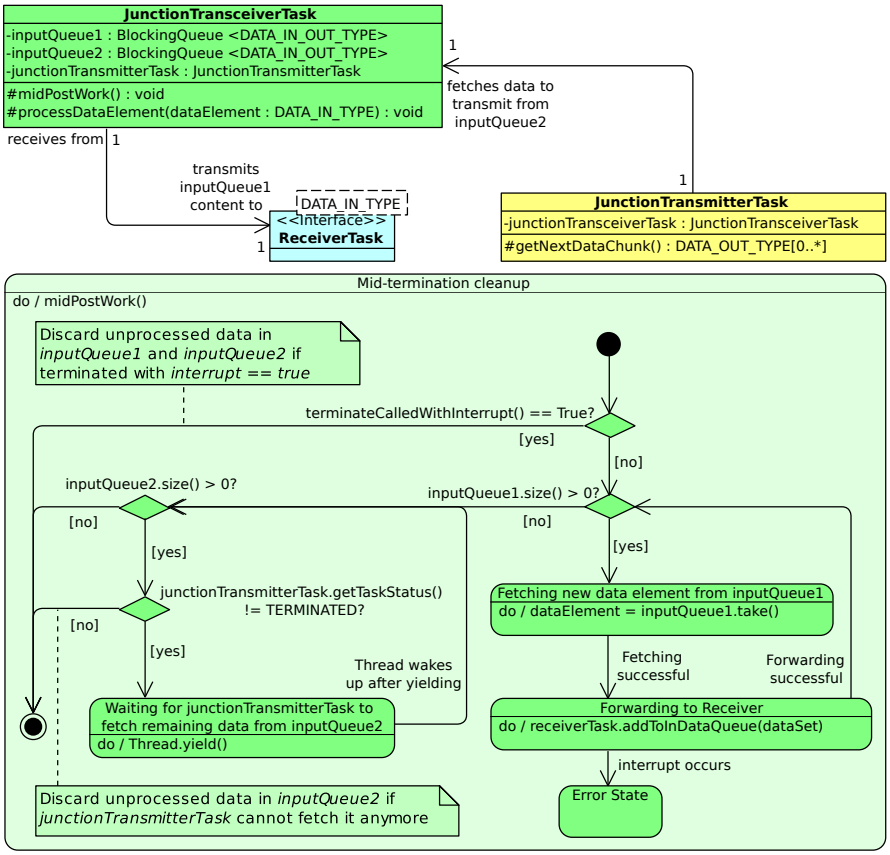

Figure 2.16: UML 2.0 state diagram depicting the lifecycle of `JunctionTransceiverTask` instances.

Figure 2.16 shows a state diagram of a `JunctionTransceiverTask` instance's `midPostWork()` method: If the latter was interrupt-terminated, then the data from both queues is discarded. Otherwise, the `JunctionTransceiverTask` instance first forwards all the remaining data in `inputQueue1` to the receiver to which it is connected. After that is done, the it waits for the associated `JunctionTransmitterTask` instance to fetch the remaining data in `inputQueue2`. If the latter terminates during that time, the former simply discards the remaining elements.

Example

Figure 2.17 depicts an example configuration containing instances of all classes involved when using a `GenericJunctionMultiTransceiverTask` instance.

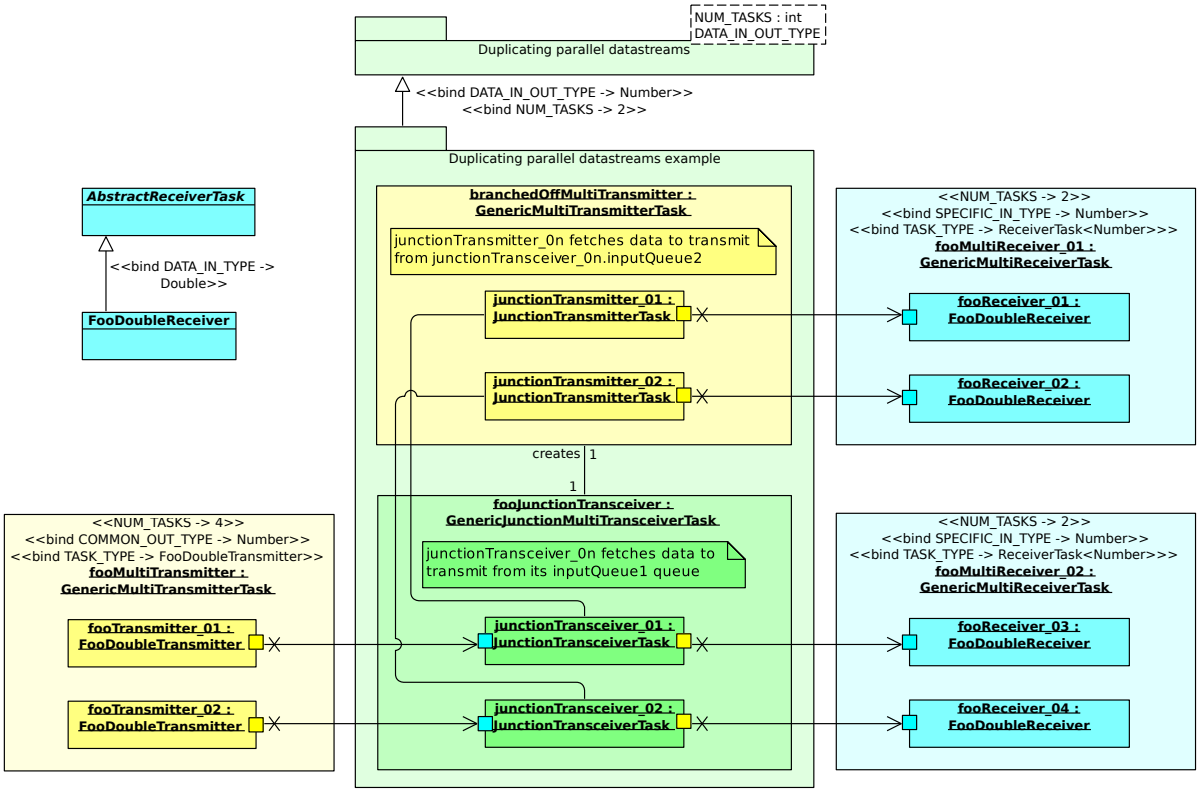

Figure 2.17: UML 2.0 object diagram depicting an example configuration employing a `GenericJunctionMultiTransceiverTask` instance.

2.6.2 Demultiplexer: From single source transmitter to multiple output transmitters

The `GenericDemuxMultiTransmitterTask` forwards data coming from a *single*, user provided transmitter instance to one of its `<NUM_TASKS>` output transmitters - thereby effectively realizing a demultiplexer. Figure 2.18 depicts its internal and dependent classes:

- `GenericDemuxMultiTransmitterTask`: In its role as a `MultiTransmitterTask`, it manages `<NUM_TASKS>` instances of `DemuxTransmitterTasks`.
- `DemuxTransmitterTask`: Forwards data from its `outputQueue`.
- `AbstractSourceTransmitterTask`: An instance of this class must be provided to the constructor of `GenericDemuxMultiTransmitterTask`. This instance represents the actual data source. It contains two methods:
  - `getNextDataChunk()`:
    1. `For(` $n = 1, \ldots,$`NUM_TASKS )`:
       1.1. `payload = getNextDataChunk();`
       1.2. `For(` $m = 1, \ldots,$`payload.size() ):`
            `dataSet[m] = Pair.of(`n`, payload[`m`])`
    2. `return dataSet;`
  - `getNextDataChunkForDemux(demuxIndex)`: An abstract method that has to be overridden by implementing subclasses. Must return either `null` or a collection of

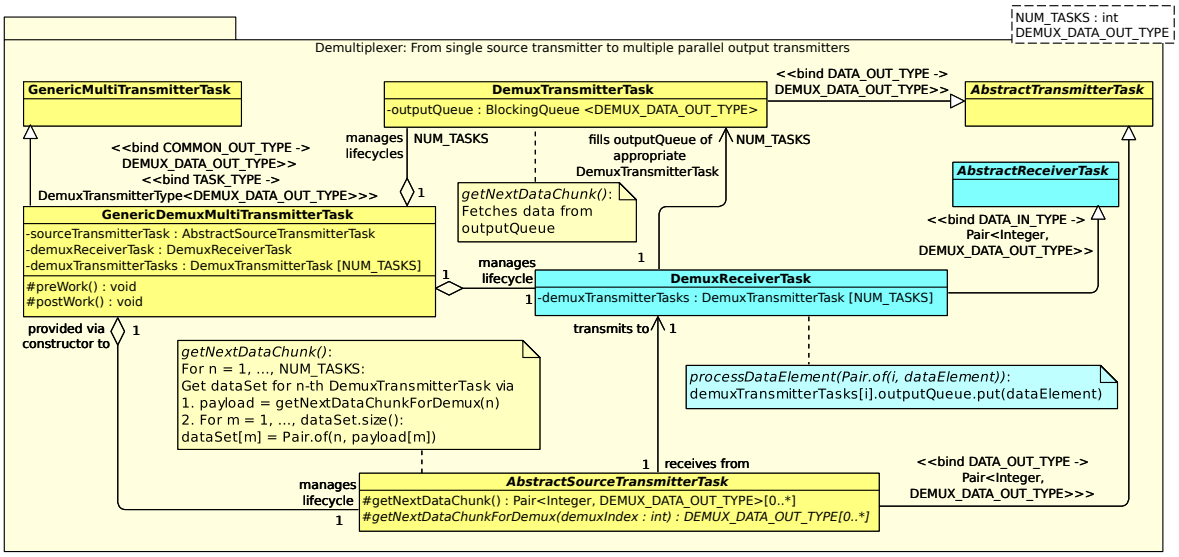

Figure 2.18: UML 2.0 class diagram depicting a demultiplexer.

data elements that shall be forwarded to the `outputQueue` of the `<demuxIndex>`-th `DemuxTransmitterTask`.

- `DemuxReceiverTask`: The `AbstractSourceTransmitterTask` instance forwards its data to this receiver. The latter serves as a distributor. Upon receiving a data element `Pair.of(`n`, payload)`, it forwards `payload` to the n-th internal `DemuxTransmitterTask`'s `outputQueue`.

Lifecycle

`GenericDemuxMultiTransmitterTask` overrides `preWork()` and `postWork()` in order to ensure the start and termination order depicted in Figure 2.19.

- `preWork()`: First start the internal `AbstractSourceTransmitterTask` instance. After it has left its `NOT_STARTED` state, start the `DemuxTransmitterTask`. When the latter is also running, call `super.preWork()`. This implicitly starts the `<NUM_TASKS>` instances of `DemuxTransmitterTask` that the `GenericDemuxMultiTransmitterTask` manages in its role as a `MultiTransmitterTask`.
- `postWork()`: Terminate the internal tasks in the same order they were started.

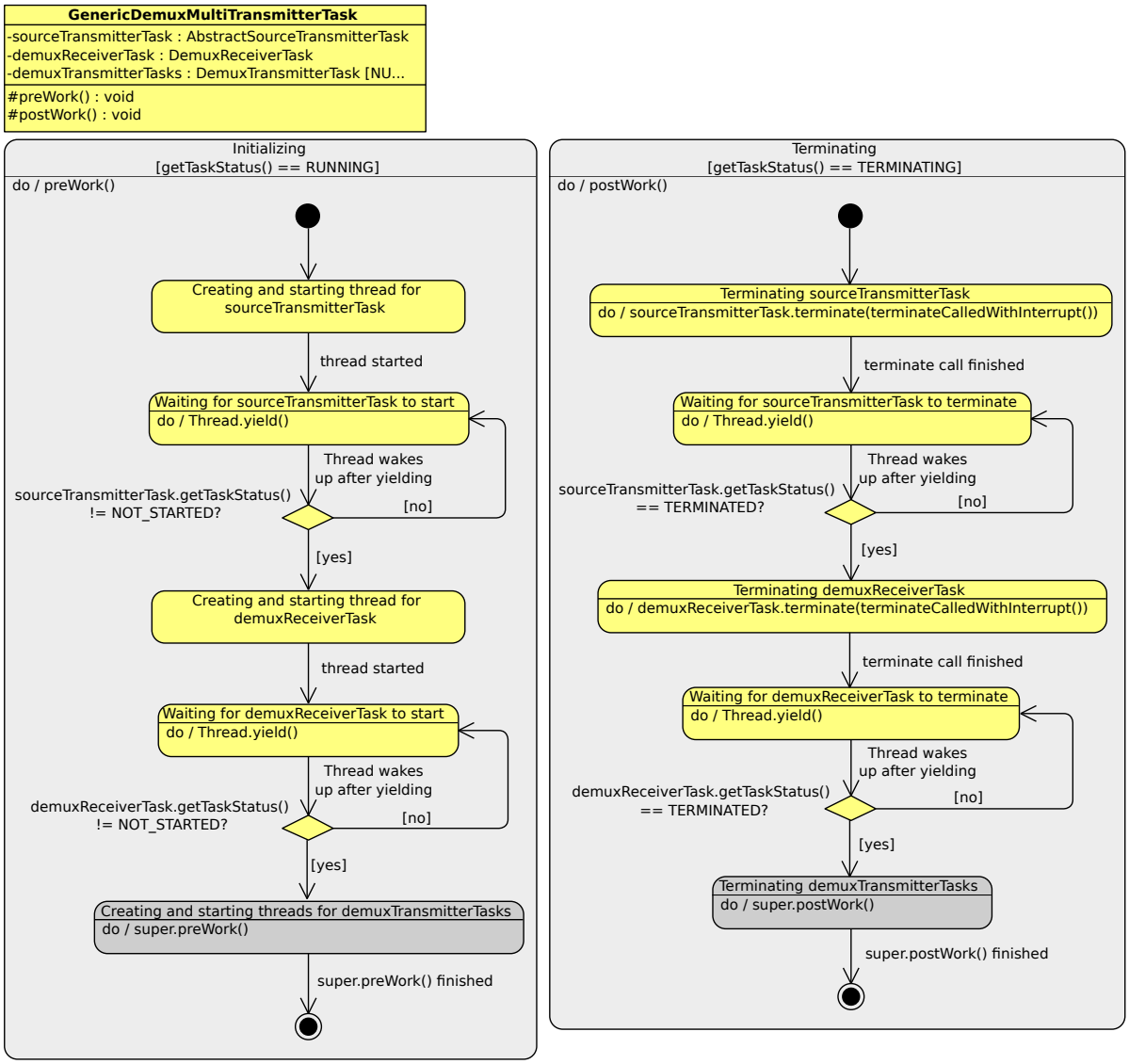

Figure 2.19: UML 2.0 state diagram depicting the lifecycle of `GenericDemuxMultiTransmitterTask` instances.

Example

Figure 2.20 depicts an example configuration containing instances of all classes involved when using a `GenericDemuxMultiTransmitterTask`.

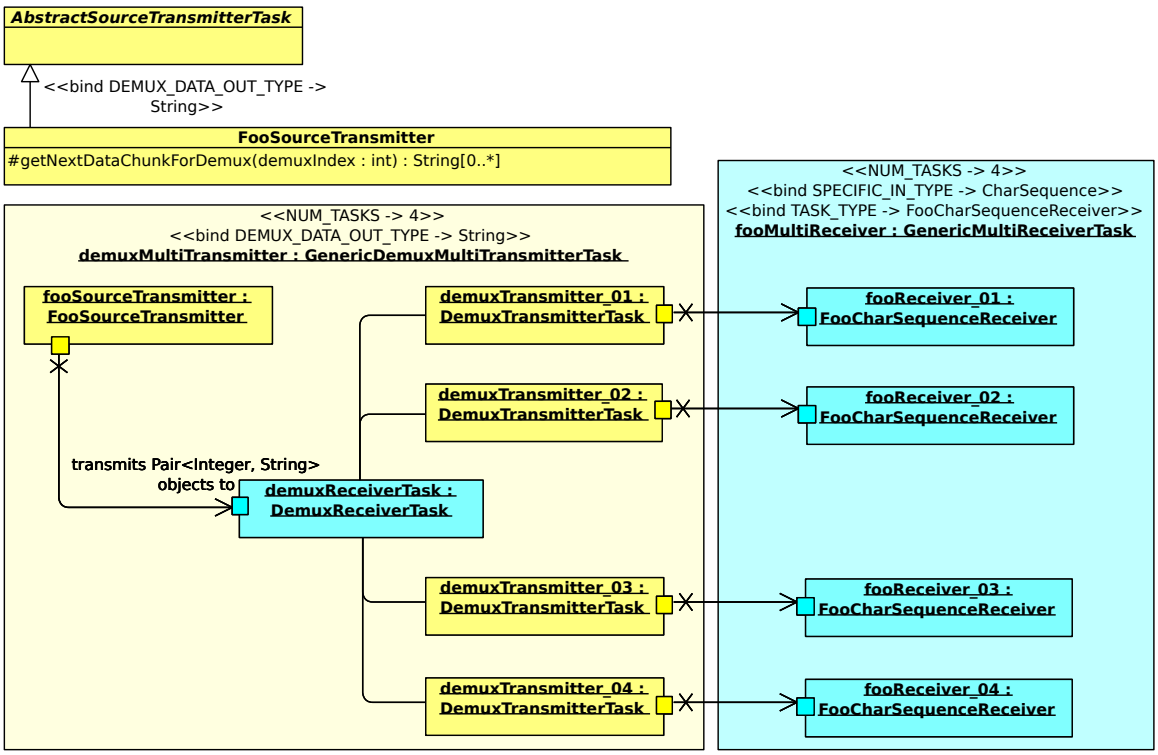

Figure 2.20: UML 2.0 object diagram depicting an example configuration employing a `GenericDemuxMultiTransmitterTask` instance.

2.6.3 Multiplexer: From multiple input receivers to single target receiver

he `GenericMuxMultiReceiverTask` forwards data coming from each of its `<NUM_TASKS>` input receivers to a *single* user provided receiver instance - thereby effectively realizing a multiplexer. Figure 2.21 depicts its internal and dependent classes:

- `GenericMuxMultiReceiverTask`: As a `MultiReceiverTask`, it manages `<NUM_TASKS>` instances of `MuxReceiverTask`.
- `MuxReceiverTask`: Puts the incoming data elements in its `inputQueue`.
- `AbstractMuxTransmitterTask`: Fetches the next data element from the `inputQueue` of the n-th `MuxReceiverTask` instance. Creates a tuple consisting of n and the data payload and forwards it to the `AbstractTargetReceiverTask` for processing. The order in which elements are fetched and forwarded from the `MuxReceiverTask` instances is not specified here.
- `SequentialMuxTransmitterTask`: A non-abstract `AbstractMuxTransmitterTask` subclass and the default used by `GenericMuxMultiReceiverTask`. Fetches data by simply iterating through all `MultiReceiverTask` instances and checking whether their queue contains an element. If that is the case that element is removed and forwarded to the `AbstractTargetReceiverTask` as described above. The iteration then continues with the next `MultiReceiverTask`.
- `AbstractTargetReceiverTask`: An instance of this class must be provided to the constructor of `GenericMuxMultiReceiverTask`. This instance represents the actual data

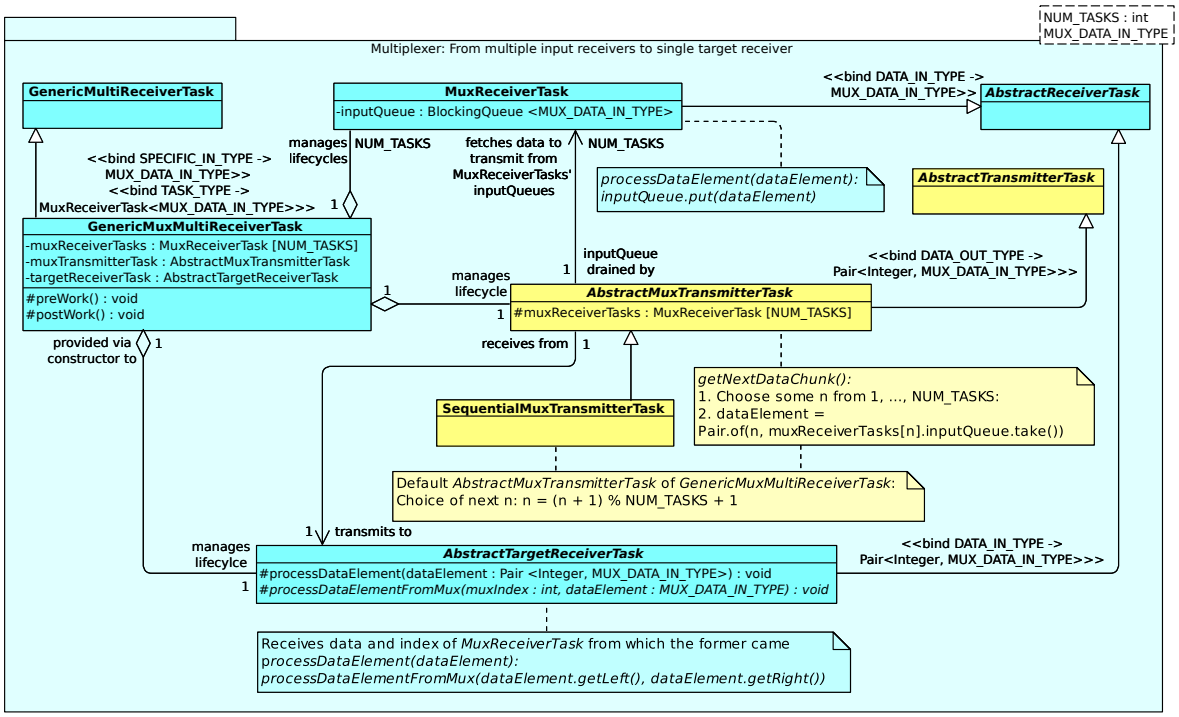

Figure 2.21: UML 2.0 class diagram depicting a multiplexer.

sink. It contains two methods:

* `processDataElement(dataElement)`: Let `dataElement = Pair.of(`n`, payload)`. Then the method simply calls `processDataElementFromMux(`n`, payload)`.
* `processDataElementFromMux(muxIndex, dataElement)`: An abstract method that has to be overridden by implementing subclasses. Must implement the application dependent data processing method. Apart from the data payload, it also provides the index of the `MuxReceiverTask` instance that received the data.

Lifecycle

`GenericMuxMultiReceiverTask` overrides `preWork()` and `postWork()` in order to ensure the start and termination order depicted in Figure 2.22:

- `preWork()`: First call `super.preWork()` in order to implicitly stars the `<NUM_TASKS>` instances of `MuxReceiverTask` that the `GenericMuxMultiReceiverTask` manages in its role as a `MultiReceiverTask`. Then start the employed `AbstractMuxTransmitterTask` instance. After it has left its `NOT_STARTED` state, start the `AbstractTargetReceiverTask` instance.
- `postWork()`: Terminate the internal tasks in the same order they were started.

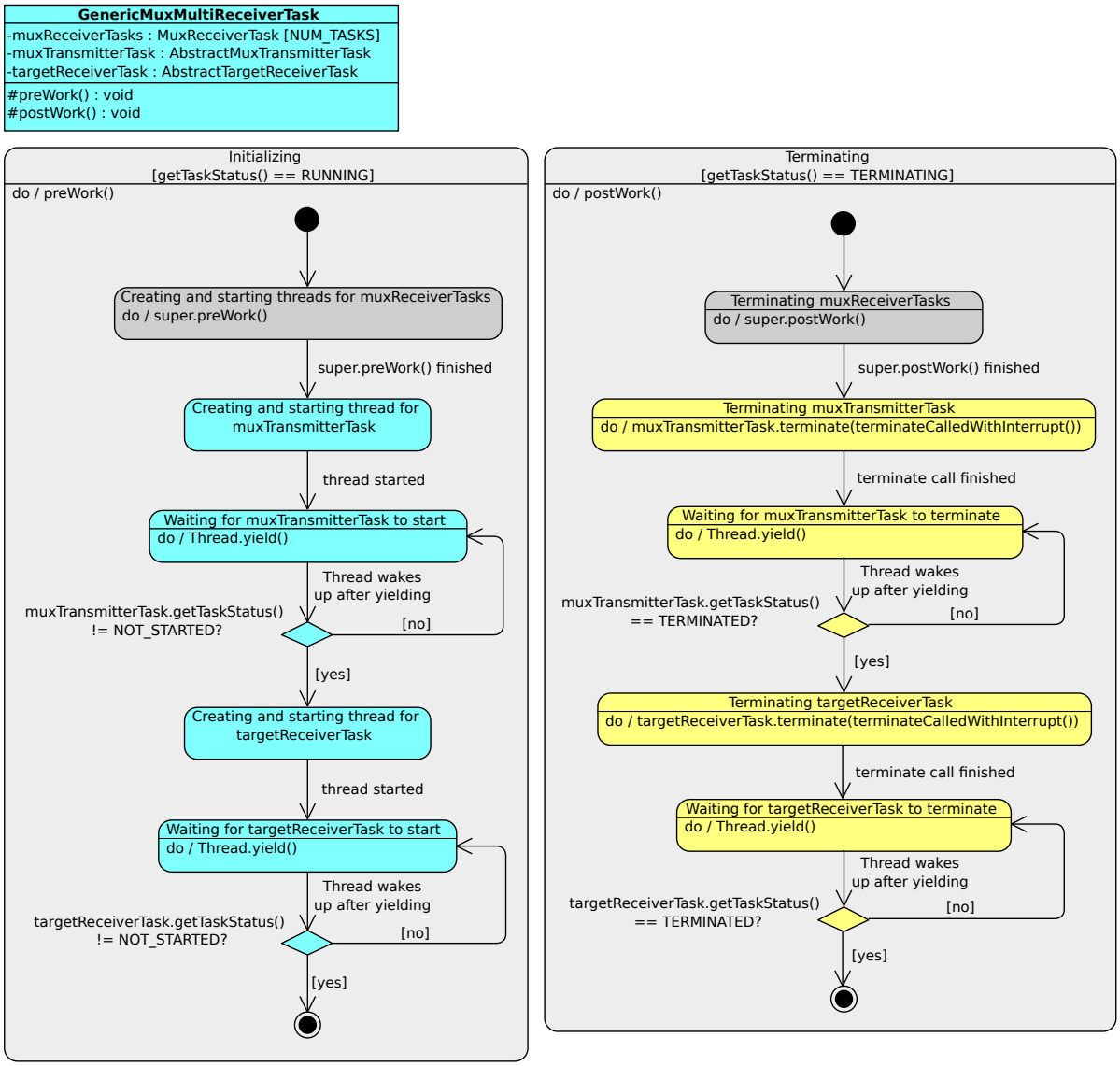

Figure 2.22: UML 2.0 state diagram depicting the lifecycle of `GenericMuxMultiReceiverTask` instances.

Example

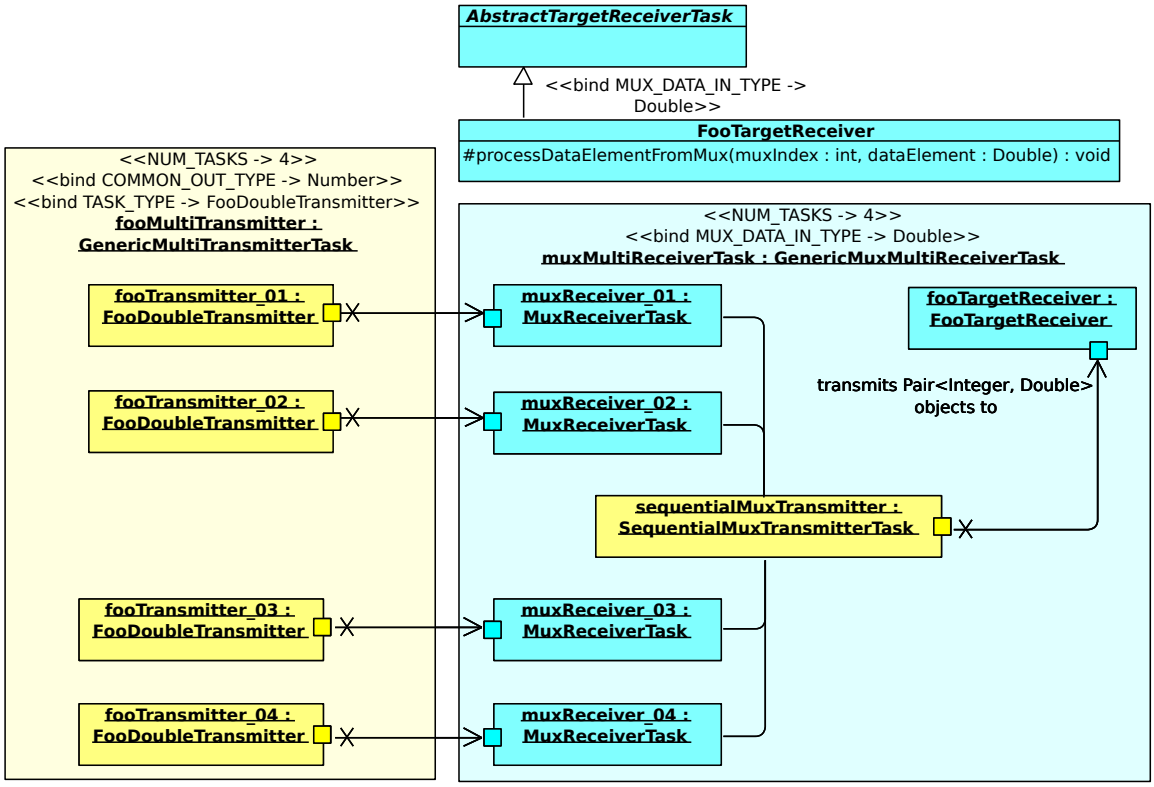

Figure 2.23: UML 2.0 diagram depicting an example configuration employing a `GenericMuxMultiReceiverTask` instance.

Figure 2.23 depicts an example configuration containing instances of all classes involved when using a `GenericMuxMultiReceiverTask`.

Ordering multiplexer

The ordering multiplexer `GenericSyncMuxReceiverTask` depicted in Figure 2.24 is a subclass of `GenericMuxMultiReceiverTask` that uses an instance of `SyncMuxTransmitterTask` instead of a `SequentialMuxTransmitterTask` in order to forward the incoming data to the `AbstractTargetReceiverTask` instance that processes it. There are several differences:

- The multiplexer input type `MUX_DATA_IN_TYPE` must now also implement the JAVA interface `Comparable<MUX_DATA_IN_TYPE>` which means that the data type must have a natural ordering (like numbers or strings).
- The data arriving at each `MuxReceiverTask` must be ordered in an ascending manner (the framework produces an error, if it detects that this is not the case).
- The `SyncMuxTransmitterTask` merges each dimension-wise ordered input data stream into an ordered single stream and forwards the it to the `AbstractTargetReceiverTask`. Note that this might involve some delays, since the `SyncMuxTransmitterTask` must wait for the `inputQueue` of each `MuxReceiverTask` instance to contain at least one element

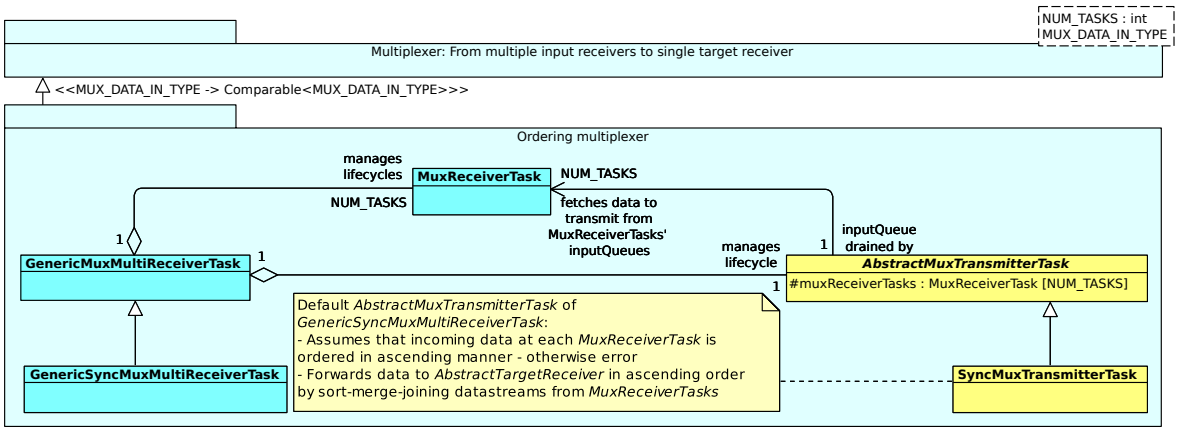

Figure 2.24: UML 2.0 class diagram depicting an ordering multiplexer that sorts its incoming data before forwarding it to its target receiver instance.

in order to guarantee, that the resulting single stream will be ordered in an ascending manner.

This concludes our documentation of the transceiver framework and we continue with a documentation of the motif detector predictor.

3 MDP: Motif detector & predictor

The *Motif Detector and Predictor (MDP)* is a data analysis tool build upon the transceiver framework from the last chapter. MPD continuously searches for motifs in each numeric input stream and outputs a string stream that denotes which input value was classified as a motif and which was not. This string stream is then forwarded, together with the symbolic input stream, to a language model trainer which continuously trains and evaluates a language model based on the input data and the ground truth. Both are provided by an instance of our *Synthetic Datastream Generator (SGD)* [10].

3.1 Color convention

We extend the color convention from the last chapter by the colors listed in Table 3.1:

Color:	Refers to:
Orange	Interfaces, classes and packages related to MDP data elements and R language models.
Magenta	Interfaces, classes and packages related to motif search.

Table 3.1: Our UML color convention.

3.2 Data types

Figure 3.1 depicts an overview on how MDP represents data: The right side contains the interface hierarchy and the right side the corresponding implementations. Our two main datatypes are `StringData` and `DoubleData`. As their name suggests, an instance of the former has a `String` object as payload and one of the latter a `Double` payload. Both data types also extend the `TimeComparable` interface. Its implementation induces an ordering based on the timestamp. By implementing `LabeledTimestampedData`, their payload can be accessed via `getData()` and their label (if present) via `getLabel()`.

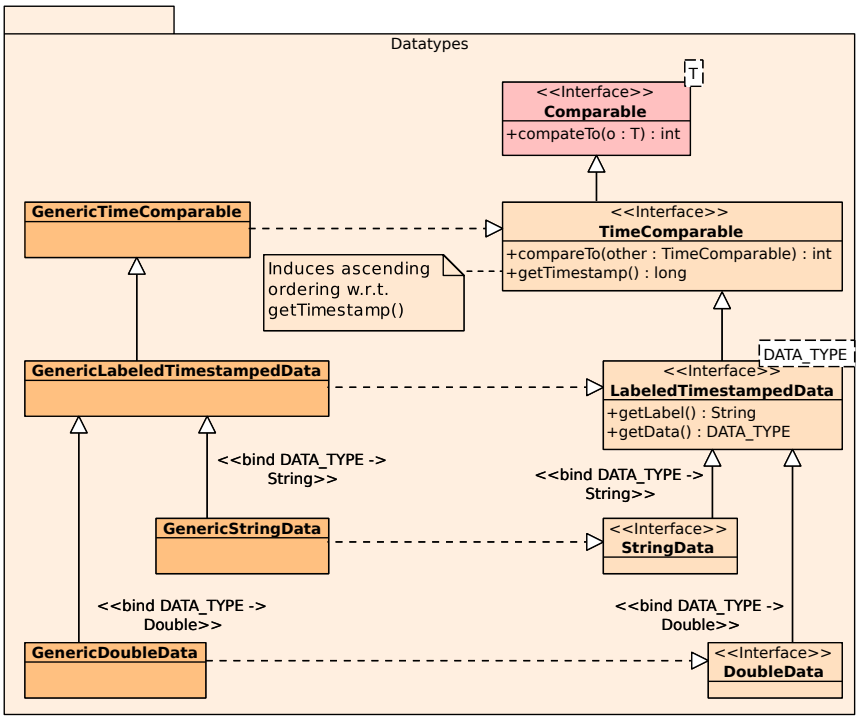

Figure 3.1: UML 2.0 class diagram depicting how MDP represents data.

3.3 SDG demultiplexer

The `SyntheticDatastreamTransmitter` is an `AbstractSourceTransmitter` implementation used in the context of `GenericDemuxMultiTransmitterTask` instances (see Section 2.6.2). It communicates with an R server using the RServe library[12] and creates an instance `synth_ds` of the `Synthetic_Datastream_Iterators` class from the SDG [10].

`SyntheticDatastreamTransmitter` implements `getNextDataChunkForDemux(demuxIndex)` by fetching new data from the `<demuxIndex>`-th dimension of the synthetic data stream instance (i.e. via `synth_ds`'s `get_next(demuxIndex)` method). Depending on whether `demuxIndex` corresponds to a numeric or a string data stream, the `SyntheticDatastreamTransmitter` parses and transforms the fetched data into `DoubleData` or `StringData` instances. The data then gets forwarded to `GenericDemuxMultiTransmitterTask`'s corresponding `DemuxTransmitterTask` task.

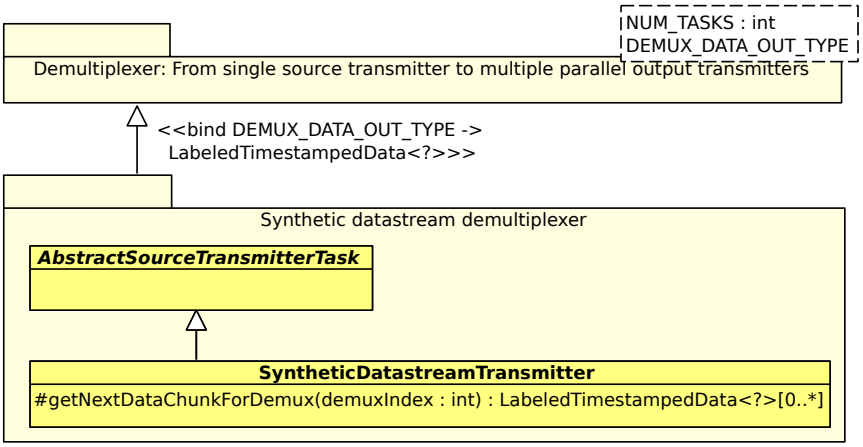

Figure 3.2: UML 2.0 class diagram depicting a demultiplexer interface to the SDG.

3.4 Motif detector transceiver

The `MotifDetectorTransceiver` depicted in Figure 3.3 receives a continuous stream of instances of `DoubleData` which it scans for for motifs. For each received `DoubleData` instance, the transceiver emits a `String` data object that holds a string payload containing information on whether the corresponding double value is part of a found motif or not. The motif search itself is performed by an instance of `MotifSearch`. This interface has the following contract:

Definition 3.1 (`MotifSearch`)**.**
A `MotifSearch` instance must implement the following methods:

- `findMotifs(dataVector)`: Receivers a double array `dataVector` and returns a set of motif candidates. Each motif candidate consists of a triple (s, l, b) where s denotes the motif candidate's start index in the data vector, l denotes the motif length and b is a "badness" value where lower values represent a "better" motif with respect to some, implementation dependent measure.
- `mergeMotifCandidates(candidates)`: Is a static method that merges the received candidate set as follows:
  1. Partition `candidates` into overlap sets. An overlap set is defined as follows: Let each candidate represent a vertex in a graph. Let two vertices be connected by an edge if the candidates overlap in the `dataVector`. Then the overlap sets correspond to the connected components of said graph.
  2. For each overlap set, choose the candidate with the lowest badness value. If this choice is ambiguous (i.e., multiple candidates have that same b value), choose the longest one. If there are still ambiguities in the choice, choose the oldest one (i.e. the one with the lowest start index).
  3. Return only those chosen candidates, sorted with respect to their start value in an ascending manner.

An instance of `MotifDetectorTransceiver` behaves as follows:

Definition 3.2 (`MotifDetectorTransceiver`)**.**
A let `mdt` be an instance of `MotifDetectorTransceiver`. `mdt` has the following fields and

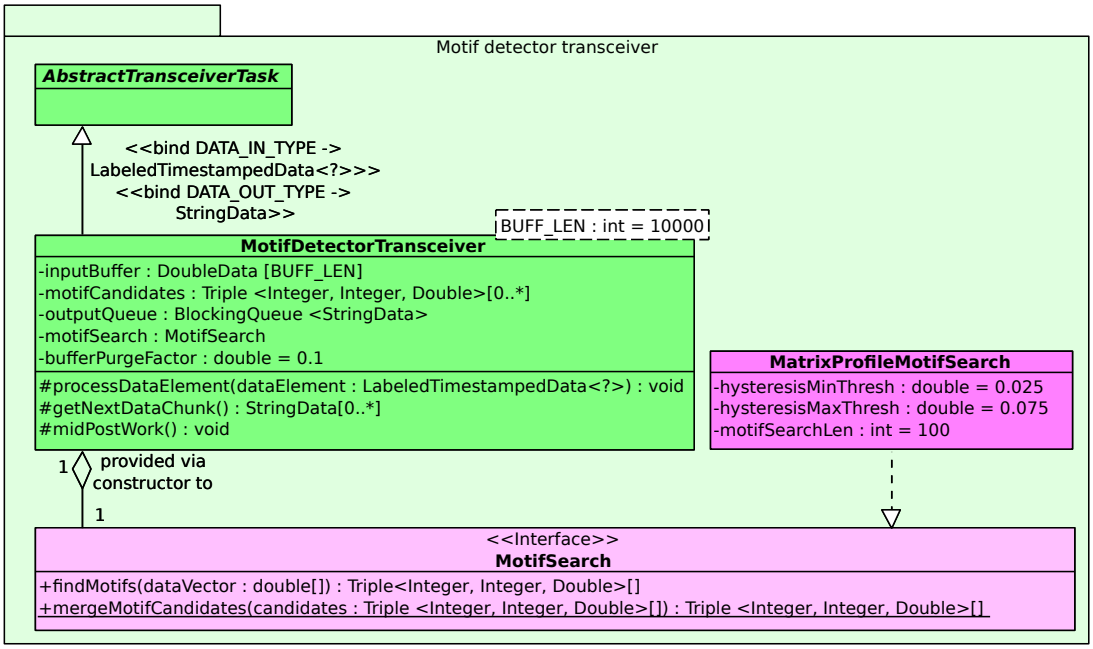

Figure 3.3: UML 2.0 class diagram depicting MPD's motif detection transceiver.

methods:

- `mdt.outputQueue`: An output queue with data that can immediately be forwarded.
- `mdt.getNextDataChunk()`: Forwards elements from the `outputQueue`.
- `mdt.processDataElement(dataElement)`: At this point, `dataElement` is guaranteed to be of type `DoubleData`. The method itself performs the following steps:
  1. Append `dataElement` to **back** of `inputBuffer`;
  2. `If( mdt.inputBuffer.size() == BUFF_LEN )`: Each time the buffer is full, perform a motif search in the whole buffer.
     2.1. `newCandidates = mdt.motifSearch .findMotives(toDoubleArray(inputBuffer))`
     `toDoubleArray(...)` simply transforms the `DoubleData` buffer content into a simple `double[]` array.
     2.2. `mdt.motifCandidates = motifSearch.mergeMotifCandidates( UNION(mdt.motifCandidates, newCandidates));`
     Update the motif candidates by merging the union of both new and old candidates.
     2.3. `purgeIndex` $= \lceil$`mdt.bufferPurgeFactor` $\cdot$ `BUFF_LEN`$\rceil$;
     2.4. Let `outputCandidates` be all the motif candidates in `mdt.motifCandidates` with a start index $\leq$ `purgeIndex`.
     2.5. `mdt.outputQueue.addAll(toStringData(outputCandidates);`
     `toStringData(...)` transforms the sequence covered by the output candidates into a sequence of `StringData` instances. The length of that sequence

equals the maximal end index of all `outputCandidates`. Each instance carries a string payload classifying it as either being the start of a detected motif, being inside a detected motif or not being part of a detected motif. Its label is unchanged and carries the ground truth label from the SDG.

2.6. `mdt.motifCanidates.remove(outputCandidates);`

2.7. Remove all entries from `mdt.inputBuffer` corresponding to the sequence covered by `outputCandidates`, i.e. all indices between 1 and the maximal end index among all `outputCandidates`.

This setup allows to easily experiment with different `MotifSearch` implementations.

3.4.1 Matrix profile based motif search

The `MotifSearch` implementation we employ is the `MatrixProfileMotifSearch` class. It relies on the concept of matrix profiles which was introduced in [13]:

Definition 3.3 (Matrix profile)**.** Let $x_1^N = (x_1, \ldots, x_N) \in \mathbb{R}^N$ be a sequence of numeric elements of length $N \in \mathbb{N}$. Let for $i \leq j$, $x_i^j := (x_i, x_{i+1}, \ldots, x_j)$ denote a subsequence in x_1^N. Furthermore, let $m \in \{1, \ldots, N\}$ be the fixed length of motifs we want to detect. We further define for two vectors $x, y \in \mathbb{R}^D$:

- Let μ_x and s_x denote the mean and standard deviation of the entries in x.
- Let $\hat{x} := \frac{x - \mu_x}{s_x}$ denote the z-normalized x.
- The z-normaized Euclidean distance $dist_z(x, y)$ is defined the Euclidean distance between $\hat{x}$ and $\hat{y}$:
$$dist_z(x, y) = \sqrt{(\hat{x} - \hat{y})^t (\hat{x} - \hat{y})}$$
- It can easily be shown that the following holds[13]:
$$dist_z(x, y) = \sqrt{2D \left(1 - \frac{x^t y - D\mu_x \mu_y}{D s_x s_y}\right)} \tag{3.1}$$
- The matrix profile $mp \in \mathbb{R}^{N-m+1}$ is defined as follows:
$$\forall i = 1, \ldots, N - m + 1 \colon mp[i] := \min_{\substack{j=1,\ldots,N-m+1 \\ j \notin [i-m, i+m]}} dist_z(x_i^{i+m-1}, x_j^{j+m-1})$$

Minima in mp therefore correspond to potential motif start indices. Note that the matrix profile as defined here is not length-invariant due to the $\sqrt{2D}$ factor in Equation (3.1). This means that the absolute values in two matrix profiles computed with different m values are not directly comparable. Instead of using the matrix profile from Definition 3.3, we use a length normalized version of the matrix profile:

Definition 3.4 (Length normalized matrix profile)**.** Let $x_1^N = (x_1, \ldots, x_N) \in \mathbb{R}^N$ be a sequence of numeric elements of length $N \in \mathbb{N}$. Let for $i \leq j$, $x_i^j := (x_i, x_{i+1}, \ldots, x_j)$ denote a subsequence in x_1^N. Furthermore, let $m \in \{1, \ldots, N\}$ be the fixed length of motifs we want to detect. We further define for two vectors $x, y \in \mathbb{R}^D$:

- Let the length normalized matrix profile $\hat{mp}$ be defined as:

$$\hat{mp} := \frac{mp}{\sqrt{2D}}$$

This is equivalent to computing the matrix profile the distance function being normalized in the same manner. Note that this has the nice theoretical property that $\hat{dist_z}(x, y) := \frac{dist_z(x,y)}{\sqrt{2D}}$ is equal to $\sqrt{1 - r_{x,y}}$ with $r_{x,y}$ being the well known empiric Pearson correlation coefficient. And in fact, one of the latest matrix profile followup papers also uses this length normalized matrix profile variant.

Now we can define the `findMotifs(dataVector)` function of our `MotifSearch` implementation `MatrixProfileMotifSearch`:

Algorithm 3.1 (findMotifs(dataVector))**.**

1. Compute the length normalized matrix profile $\hat{mp}$ of the provided `dataVector` with respect to the motif search length `motifSearchLen`; We apply the method from [14] which to compute the matrix profile in $\mathcal{O}(N^2)$ with N being the data vector length. The length normalization then takes only $\mathcal{O}(N)$.
2. Find significant minima in the matrix profile using a hysteresis like approach: Iterate from left to right through the matrix profile values. As soon as a value is smaller than `hysteresisMinThresh`, compute the interval starting at that value and ending at the last value that is smaller than `hysteresisMaxThresh`. Generate a candidate (s, l, b) with s being set to the index of the minimal matrix profile value inside this interval, l being simply the motif search length and $b := \hat{mp}[s]$. Then continue with the search.

Note that the upper approach is currently not capable of finding motifs of varying lengths. Nonetheless, the implementation provides a framework for future research into this issue.

3.5 Continuous F1 evaluation multiplexer

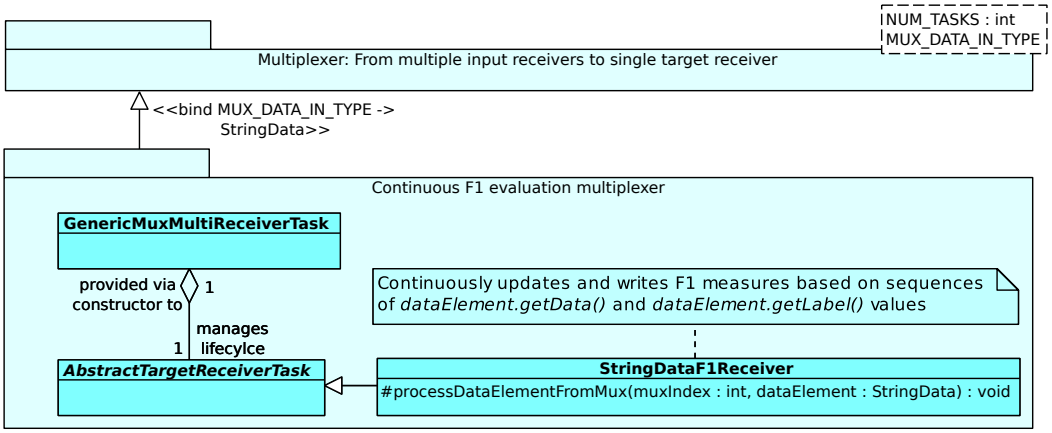

Figure 3.4: UML 2.0 class diagram depicting the F1 evaluation multiplexer .

The `StringDataF1Receiver` class is used as an `AbstractTargetReceiverTask` instance inside a `GenericMuxMultiReceiveTask` instance. In our framework, the output streams of the motif detector transceiver are forked via a junction transceiver and provided to a multiplexer

that in turn joins and forwards the stream to the `StringDataF1Receiver`. The latter then continuously computes an F1 like measure based on the found motifs and the ground truth about the "true" motifs inserted by the SGD. Our F1 variant is strongly inspired by the E4SC measure from [15] and very robust to an unequal distribution of positive and negative labels (or, in our case, NA sequences and motifs). It takes a value of 1 if and only if all motif have been correctly identified and no NA sequences have been falsely labeled as motifs. We omit a formal derivation and refer to our code at [11].

3.6 Ordering multiplexer for continuous language model training and evaluation

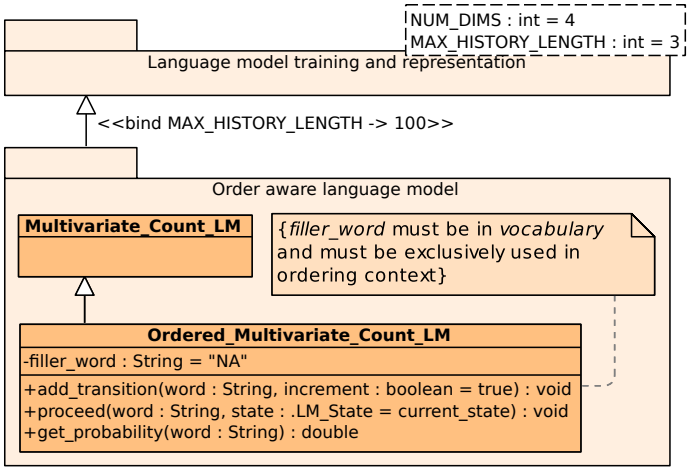

Figure 3.5: UML 2.0 class diagram depicting the employed language model R class.

Note that the motif detector transceiver is only able to tell in which subsequence of the stream it detected the motif but it is unable to further classify motifs by shape and thereby reconstruct a labeling which is bijective to the labels emitted by the underlying language model that determines the order in which SDG emits labels. In order to somewhat compensate this drawback, we extended the SDG class `Multivariate_Count_LM` by a subclass called `Ordered_Multivariate_Count_LM`. Contrary to the former, the latter's transition function is defined in a manner to preserve information on the order in which elements are added to its history:

Definition 3.5 (Ordered_Multivariate_Count_LM)**.**
Let `olm` be an instance of `Ordered_Multivariate_Count_LM`. `olm` contains the following fields and overrides the following methods (see [10]):

- `olm.filler_word`: Must be a special word that the LM only uses in the context descibed below.
- `lm.add_transition(word, increment)`:
  1. `If( word == olm.idle_word:` super.add_transition(word);
  2. `Else:`
     2.1. Let d(`word`) be the dimension of `word`;

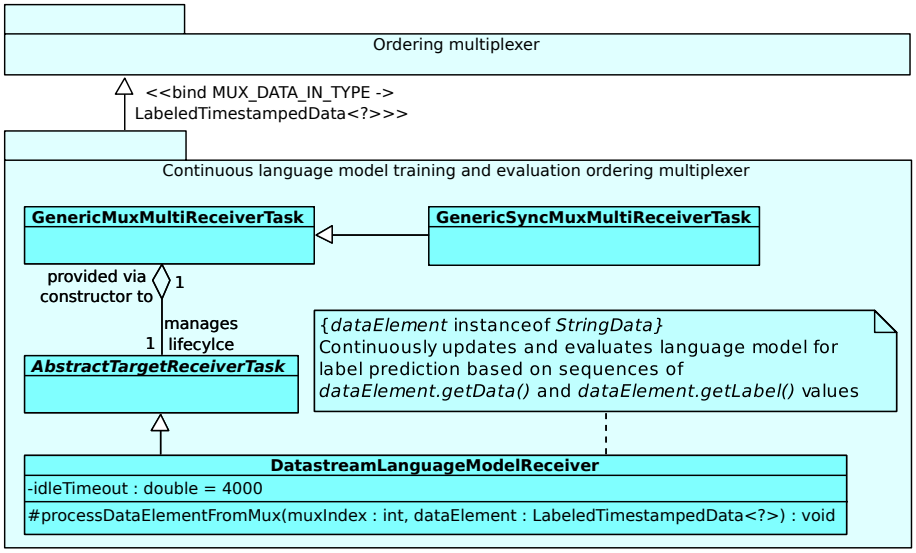

Figure 3.6: UML 2.0 class diagram depicting the language model's integration into an ordered multiplexer.

2.2. `For(` $i = 1, \ldots, d-1$ `):`
`super.add_transition(filler_word, increment);`

3. `super.add_transition(word, increment);`
4. `For(` $i = d+1, \ldots,$ `NUM_DIMS ):`
`super.add_transition(filler_word, increment);`

- `get_probability(word):`
  1. $p = 1$;
  2. `old_state = olm.current_state;`
  3. `For(` $i = 1, \ldots, d-1$ `):`
     3.1. $p = p \cdot$ `super.get_probability(olm.filler_word);`
     3.2. `super.proceed(olm.filler_word);`
  4. $p = p \cdot$ `super.get_probability(word);`
  5. `For(` $i = d+1, \ldots,$ `NUM_DIMS ):`
     5.1. $p = p \cdot$ `super.get_probability(olm.filler_word);`
     5.2. `super.proceed(olm.filler_word);`
  6. `olm.current_state = old_state;`
  7. `return` p;

An instance of this model is then managed by a `DataStreamLanguageModelReceiver` instance. The latter is part of a `GenericSyncMuxMultiReceiverTask` (Section 2.6.3). It is therefore guaranteed that the data forwarded to the former is ordered in an ascending manner with respect to their timestamps. Note that the `DataStreamLanguageModelReceiver` adds an idle transition to the language model each time no non-NA label has been encountered for the period stored in the `idleTimeout` field.

3.7 MDP example configuration

Figure 3.7 and 3.8 illustrate the set of `ConcurrentTask` (Section 2.2) instances involved in a running instance of MDP with two numeric and two symbolic dimensions.

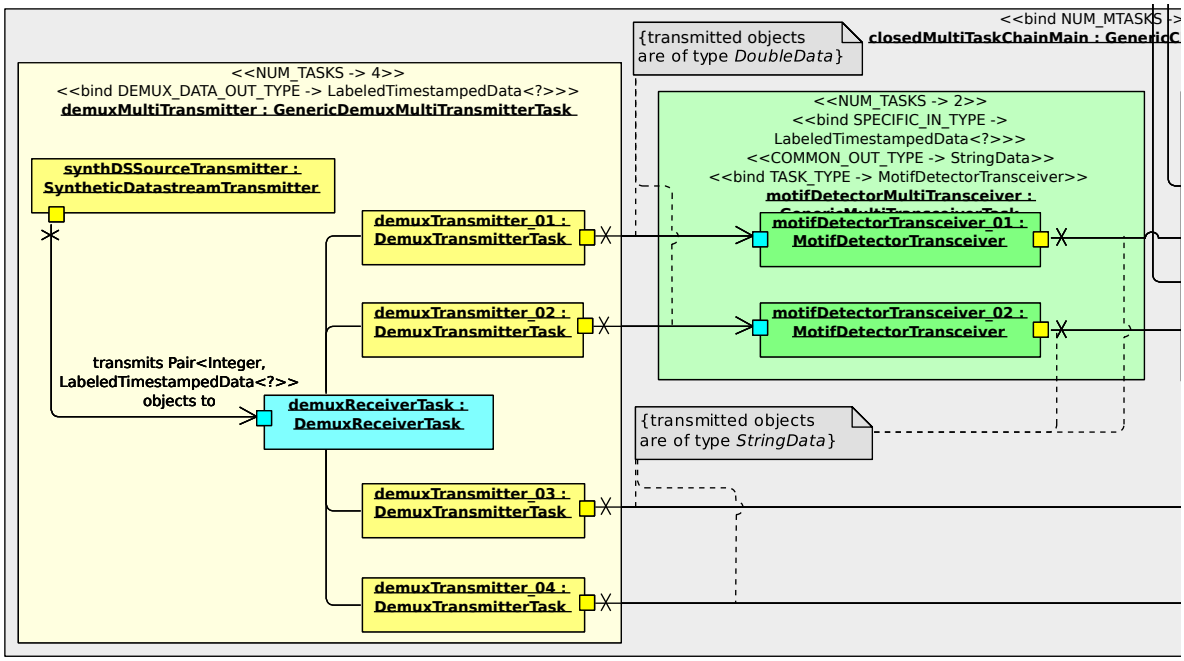

Figure 3.7: First half of an UML 2.0 object diagram depicting an example configuration of an MDP instance with two numeric and two symbolic dimensions. See Figure 3.8 for the right part.

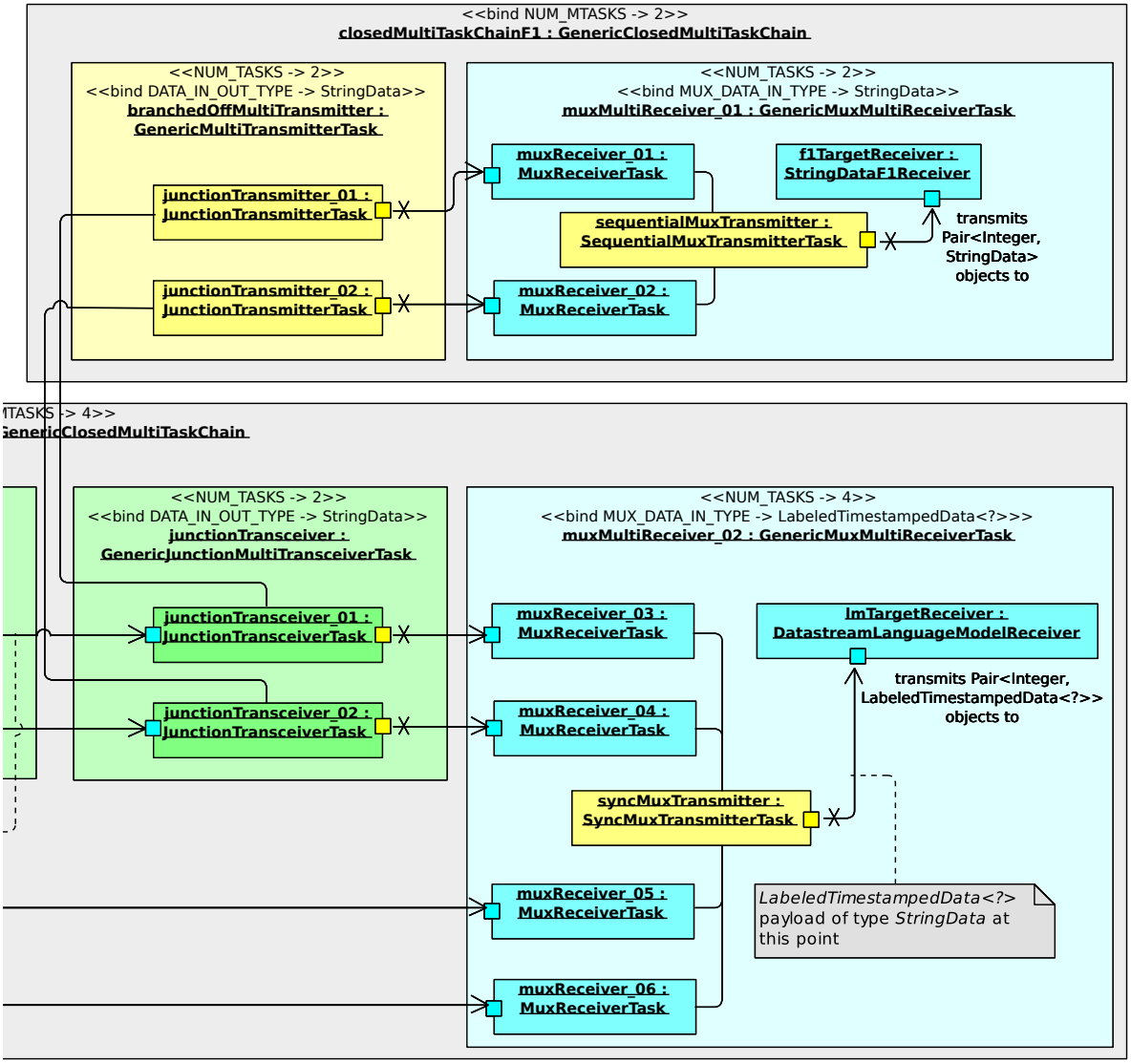

Figure 3.8: Right part of Figure 3.7

Bibliography

[1] Graciela H. Gonzalez, Tasnia Tahsin, Britton C. Goodale, Anna C. Greene, and Casey S. Greene. Recent advances and emerging applications in text and data mining for biomedical discovery. *Briefings in Bioinformatics*, 17(1):33–42, 2016. doi: 10.1093/bib/bbv087. URL http://dx.doi.org/10.1093/bib/bbv087.

[2] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5:4308 EP –, Jul 2014. URL https://doi.org/10.1038/ncomms5308. Article.

[3] A. A. Pol, G. Cerminara, C. Germain, M. Pierini, and A. Seth. Detector monitoring with artificial neural networks at the CMS experiment at the CERN Large Hadron Collider. *ArXiv e-prints*, July 2018.

[4] Q. Wang, Y. Guo, L. Yu, and P. Li. Earthquake prediction based on spatio-temporal data mining: An lstm network approach. *IEEE Transactions on Emerging Topics in Computing*, pages 1–1, 2018. ISSN 2168-6750. doi: 10.1109/TETC.2017.2699169.

[5] Devendra Kumar Tayal, Arti Jain, Surbhi Arora, Surbhi Agarwal, Tushar Gupta, and Nikhil Tyagi. Crime detection and criminal identification in india using data mining techniques. *AI & SOCIETY*, 30(1):117–127, Feb 2015. ISSN 1435-5655. doi: 10.1007/s00146-014-0539-6. URL https://doi.org/10.1007/s00146-014-0539-6.

[6] Félix J. López Iturriaga and Iván Pastor Sanz. Bankruptcy visualization and prediction using neural networks: A study of u.s. commercial banks. *Expert Systems with Applications*, 42(6):2857 – 2869, 2015. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2014.11.025. URL http://www.sciencedirect.com/science/article/pii/S0957417414007118.

[7] G. A. Susto, A. Schirru, S. Pampuri, S. McLoone, and A. Beghi. Machine learning for predictive maintenance: A multiple classifier approach. *IEEE Transactions on Industrial Informatics*, 11(3):812–820, June 2015. ISSN 1551-3203. doi: 10.1109/TII.2014.2349359.

[8] Yang Lu. Industry 4.0: A survey on technologies, applications and open research issues. *Journal of Industrial Information Integration*, 6:1 – 10, 2017. ISSN 2452-414X. doi: https://doi.org/10.1016/j.jii.2017.04.005. URL http://www.sciencedirect.com/science/article/pii/S2452414X17300043.

[9] David R. Shelly. Possible deep fault slip preceding the 2004 parkfield earthquake, inferred from detailed observations of tectonic tremor. *Geophysical Research Letters*, 36(17):n/a–n/a, 2009. ISSN 1944-8007. doi: 10.1029/2009GL039589. URL http://dx.doi.org/10.1029/2009GL039589. L17318.

[10] Synthetic data steam generator. URL https://github.com/GStepien/SDG. [last accessed December 21, 2018].

[11] Transceiver framework. URL `https://github.com/GStepien/Transceiver_Framework`. [last accessed December 21, 2018].

[12] Rserve. URL `https://www.rforge.net/Rserve/`. [last accessed December 21, 2018].

[13] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn Keogh. Matrix profile i: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pages 1317–1322. IEEE, 2016.

[14] Yan Zhu, Zachary Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth Funning, Abdullah Mueen, Philip Brisk, and Eamonn Keogh. Matrix profile ii: Exploiting a novel algorithm and gpus to break the one hundred million barrier for time series motifs and joins. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pages 739–748. IEEE, 2016.

[15] Stephan Günnemann, Ines Färber, Emmanuel Müller, Ira Assent, and Thomas Seidl. External evaluation measures for subspace clustering. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 1363–1372, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0717-8. doi: 10.1145/2063576.2063774. URL `http://doi.acm.org/10.1145/2063576.2063774`.