CHAPTER 45

How to Write Your Own Extensions and Possibly Contribute Them to MATSim

Michael Zilske

Notes

Documentation for the concepts described in this chapter can be found under `http://matsim.org/javadoc` → main distribution, by going to the corresponding class and interface documentation entries. These should also point to examples.

For programming against the MATSim API, we recommend `https://github.com/matsim-org/matsim-example-project` as a starting point; in particular, this should clarify how MATSim can be used as an Apache Maven plug-in.

45.1 Introduction

The three main elements of the MATSim cycle, execution, scoring, and replanning (Section 1.4), operate on what is essentially an in-memory, object-oriented data base of `Person` objects (Raney and Nagel, 2006). These three elements are the main elements to configure MATSim:

Execution The mobsim can be replaced, either by an internally available alternative, or by a fully external mobsim.

Scoring The scoring can be replaced, by possibly giving each individual agent a different recipe to compute its score.

Replanning Arbitrary implementations of type `PlanStrategy` can be added to the replanning; these either generate new plans from scratch or mutate existing ones, or they select between plans.

The simulation's behavior can be further configured by using `ControlerListeners`.

How to cite this book chapter:

Zilske, M. 2016. How to Write Your Own Extensions and Possibly Contribute Them to MATSim. In: Horni, A, Nagel, K and Axhausen, K W. (eds.) *The Multi-Agent Transport Simulation MATSim*, Pp. 297–304. London: Ubiquity Press. DOI: http://dx.doi.org/10.5334/baw.45. License: CC-BY 4.0

The mobsim generates a stream of events. These are primarily used in two places:

- The scoring uses events to track each agent's success at executing its plan, and computes the scoring value based on this.
- `PlanStrategy` modules use events to build approximate models of the world in which they operate. For example, the router obtains time-dependent expected link travel times from a `TravelTime` object, which in turn listens to link enter and link exit events.

Additionally, one can write any sort of event handlers for analysis during the iterations or after a run by evaluating the events file.

Some modules are so large that fully replacing them in order to adapt the simulation system to one's needs is too much work. These are, in particular,

- the QSim, which is the default implementation of the `Mobsim` interface, and
- the router.

As a result, it is possible to add additional executable code into the execution flow of the QSim by `MobsimListeners` in a similar way as this is possible with the `ControlerListeners` mentioned above. The router, in contrast, is most importantly configured by replacing the definition of the generalized travel cost.

45.2 Extension Points

This section describes what could be called the SPI (Service Provider Interface) of MATSim. Historically, the main entry-point for writing a MATSim extension has been to literally extend (in the Java sense, i.e. to inherit from) the `Controler` class. Essentially, one would override the methods calling the mobsim, the scoring, and/or the replanning, as explained in Section 45.1. This is now discouraged. While this pattern worked when a each member of the team was working on extending the MATSim core by a different aspect, it fails when it comes to integrating those aspects to a single product: There is nothing one can do with a `PublicTransportControler`, an `EmissionsControler`, a `RoadPricingControler` and an `OTFVisControler`, if one wants to combine them to visualize the emissions of buses on toll roads. Also see Section 46.2.1.5.

45.2.1 Config Group

The configuration of a MATSim run is a grouped list of key-value pairs, stored in XML format in the config file (see Section 45.2.1).

At runtime, the entire configuration is stored in an instance of `Config`, from which instances of `ConfigGroup` can be accessed by their name. Config groups that are not in the main distribution need to be explicitly loaded; an approximate example is the following:

```
MyExternalConfigGroup myConfig
   = ConfigUtils.addOrGetModule(controler.getConfig(),
      MyExternalConfigGroup.GROUP_NAME,
      MyExternalConfigGroup.class);
```

The author of an extension can subclass the `ConfigGroup` class to provide named accessors for the parameters. A possibly better way is to subclass from `ReflectiveConfigGroup`, which you can use if you want to define the mapping of named parameters to accessors using Java annotations.

See `http://matsim.org/javadoc` → main distribution → `RunReflectiveConfigGroup` for an example.

45.2.2 *ObjectAttributes and Customizable*

MATSim operates on data types such as links, nodes, persons, or vehicles. Many of these data types have attributes, such as free speed (for links) or coordinates (for nodes). Rather often, one would like additional information for certain data types, such as "slope" for links, or "age" for persons. In order to not modify the data types every time this becomes necessary, but still allow experimentation, a helper container called `ObjectAttributes` is available. It essentially attaches arbitrary additional information to objects *that have an ID*, by a syntax of type

```
attribs.putAttribute( id, attribName, attribValue ) ;
```

where `id` is the object's ID, `attribName` is the name (type) of the attribute to be stored (e.g., "age"), and `attribValue` is the value of the attribute (e.g., "24").

Importantly, the package provides readers and writers for such attributes. It is thus possible for additional code to, say, generate additional attributes by preprocessing, write them to file, and read them back for every run. That approach is used, for example, by the Gauteng scenario (Chapter 69) to pre-allocate e-tag ownership to persons.

Note that there is currently no simple way to similarly attach information to data types that do not have an ID. This is, for example, the case with plans, activities, or legs, which are contained inside a data type with an ID (the person data type), but which do not possess an ID of their own and are therefore not addressable by `ObjectAttributes`. Some of these non-identifiable data types implement the Java interface `Customizable`, to which additional material can be attached by a syntax of type

```
plan.getCustomAttributes.put("myAttribName",myAttribValue) ;
```

For additional information, see the `Customizable` interface under `http://matsim.org/javadoc` → main distribution. Note that information contained in `Customizable` is not considered standard information by MATSim. It is not written to file when writing the corresponding container, it is in consequence not read from file, and it is undefined if it is copied when copying the data object (e.g., when cloning plans for the evolutionary algorithm). This is the status quo; the MATSim team is thinking about better solutions.

Please check the documentation of `ObjectAttributes` (see `http://matsim.org/javadoc` → main distribution) for more details and pointers to examples.

45.2.3 *Scenario Element*

The object-oriented, in-memory database which holds the `Person` objects with their plan memories is accessible via the `Population` interface. The `Network` interface gives access to the traffic network graph, consisting of links and nodes. There is a `TransitSchedule` interface which represents the public transit schedule. Your own modeling tasks may need an additional data container like these. We call them scenario elements. The freight carrier population of the freight extension described in Section 24.2 is a typical example.

`Scenario` is the interface which ties all scenario elements together. You can add your own named scenario element to the `Scenario` at startup, for example in a `StartupListener`. All standard scenario elements are populated from XML files at startup, but your own scenario elements could just as well be interfaces to an external relational database.

See `http://matsim.org/javadoc` → main distribution → `RunScenarioElementExample` for an example. Note, however, that in the meantime, the injection framework may have become a better alternative.

45.2.4 *ControlerListener: Handling Controler Events*

`Controler` remains the main user-facing class of MATSim, but please do not subclass it. Rather, use its setter methods to plug in your own code.[1]

`ControlerListeners` are called at the transitions of the MATSim loop (Figure 45.1), where so-called `ControlerEvents` are fed to the listeners.

The following `ControlerListeners` are currently available: `StartupListener`, `IterationStartsListener`, `BeforeMobsimListener`, `AfterMobsimListener`, `ScoringListener`, `IterationEndsListener`, `ReplanningListener`, and `ShutdownListener`. An up-to-date list can be obtained from `http://matsim.org/javadoc` → main distribution → `ControlerListener` interface.

A sample listener might look as follows.

```java
public class MyControlerListener implements StartupListener {
    @Override
    public void notifyStartup(StartupEvent event) {
        ...
    }
}
```

`ControlerListeners` are called in undefined order, meaning that `AControlerListener` may only rely on the computation of `BControlerListener` if `BControlerListener` makes that computation in an earlier transition. For instance, if `BControlerListener` is a `StartupListener` and loads data into a `Map` on start-up, `AControlerListener` can be an `IterationStartsListener` and use that `Map`. But do not write two `IterationStartsListeners` where the first puts some data into a `Map` and the second expects to find it there, they may be called in any order.

Please check the documentation of `ControlerListener` (see `http://matsim.org/javadoc` → main distribution) for more details and pointers to examples.

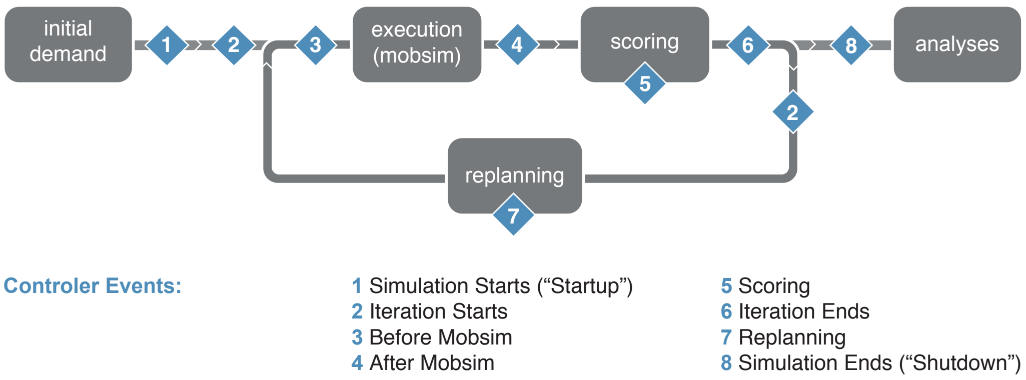

Figure 45.1: Controler events.

[1] Again, in the meantime, the injection framework may have become a better alternative altogether. The general structure, however, remains the same.

45.2.5 Events

The mobsim moves the agents around in the virtual world according to their plans and within the bounds of the simulated reality. It documents their moves by producing a stream of events. Events are small pieces of information describing the action of an object at a specific time. Examples of such events are (also see Figure 2.2):

- An agent finishes an activity.
- An agent starts a trip.
- A vehicle enters a road segment.
- A vehicle leaves a road segment.
- An agent boards a public transport vehicle.
- An agent arrives at a location.
- An agent starts an activity.

Each event has a timestamp, a type, and additional attributes required to describe the action like a vehicle id, a link id, an activity type or other data. In theory, it should be possible to replay the mobsim just by the information stored in the events. While a plan describes an agent's intention, the stream of events describes how the simulated day actually was.

As the events are so basic, the number of events generated by a mobsim can easily reach a million or more, with large simulations even generating more than a billion events. But as the events describe all the details from the execution of the plans, it is possible to extract essentially any kind of aggregated data one is interested in. Practically all analyses of MATSim simulations make use of events to calculate some data. Examples of such analyses are the average duration of an activity, average trip duration or distance, mode shares per time window, number of passengers in specific transit lines and many more.

The scoring of the executed plans makes use of events to find out how much time agents spend at activities or for traveling. Some replanning modules might make use of events as well: The router for example can use the information contained in events to figure out which links are jammed at certain times and route agents around that jam when creating new plans.

Handling Events MATSim extensions can watch the mobsim by interpreting the stream of events. This is done by implementing the `EventHandler` interface and registering the implementation with the framework. The lifecycle of an `EventHandler` can be chosen by the developer. Normally, an `EventHandler` lives as long as the simulation run. It is notified before the beginning of each new iteration so that its state can be reset to listen to a new iteration. This pattern can be used to collect information over all iterations. But if the purpose of an `EventHandler` is to make a calculation based on one single iteration, it may be more natural to create a new `EventHandler` instance for each iteration, query it for its result and discard it after the iteration finishes. This can be done in a `ControlerListener`.

See `http://matsim.org/javadoc` → main distribution → `EventHandler` for pointers to coding examples.

Producing Your Own Events One can extend the MATSim event model by extending the `Event` class to define own event types. Events can be produced from all places in the code which are executed during the running mobsim, and in particular from other `EventHandler` instances. Assume for example you want to analyze left-turns. A good starting point would be to specify a `LeftTurnEvent` class, and produce an instance of this class whenever a vehicle does a left-turn. You may do this from a class which is a `LinkLeaveEventHandler` as well as a `LinkEnterEventHandler`. A `LinkLeaveEvent` is produced every time a vehicle leaves a road segment, and a `LinkEnterEvent` is produced when it enters the next road segment. Pairing each `LinkLeaveEvent` with the next

`LinkEnterEvent` for the same vehicle gives a model for a vehicle crossing a node. At this point, your code would look at the road network model to determine if this was a left-turn, and if so, produce a `LeftTurnEvent`.

See `http://matsim.org/javadoc` → tutorial → `RunCustomScoringExample` for an example.

45.2.6 Mobsim Listener

A `MobsimListener` is called in each simulation timestep. This can, for example, be used to produce a custom event which is not triggered by another event. For example, if you wanted to include a model of weather conditions into the simulation, you could use this extension point to decide in every time step if it should start or stop raining on a certain road segment, and produce custom events for this. You would then calculate rain exposure per agent by adding an `EventHandler` which handles `LinkEnterEvent`, `LinkLeaveEvent` and your custom rain events.

Note that `EventHandler` and `MobsimListener` instances may be run in parallel by the framework. It is generally not safe to share state between them. The framework guarantees that the methods of an `EventHandler` instance are called sequentially, but two different instances may run on different threads of execution. Access to shared data must be synchronized externally. Whenever possible, different `EventHandler` instances should only communicate through events.

Example See `http://matsim.org/javadoc` → main distribution → `MobsimListener` for more details and pointers to examples.

45.2.7 TripRouter

A `TripRouter` is a service object providing methods to generate *trips* between locations, given a (main) mode, a departure time and a `Person`. A trip is a sequence of plan elements representing a movement. It typically consists of a single leg (`Leg` object), or of a sequence of legs with "*stage activities*" in between. For instance, public transport trips contain `pt interaction` activities, representing changes of vehicles in public transport trips.

Using the Router A `TripRouter` instance provides a few methods to work with trips, namely

- compute a route for a given mode and O-D pair, for a `Person` with a specific departure time,
- identify the *main mode* of a trip. For instance, a trip composed of several *walk* and *pt* legs should be identified as a public transport trip.
- Identify which activities are *stage activities*, and, by extension, identify the trips in the plan: A trip is the longest sequence of consecutive `Legs` and *stage activities*

Please check the documentation of the `TripRouter` class (see `http://matsim.org/javadoc` → main distribution) for more details and pointers to examples.

Configuring the Router The `TripRouter` computes routes by means of `RoutingModule` instances, one of which is associated with each mode. A `RoutingModule` defines the way a trip is computed, and is able to identify the *stage activities* it generates.

The association between modes and `RoutingModule` instances is configurable. You can even provide you own `RoutingModule` implementations. Do this if your use-case requires custom routing logic, for instance, if you want to implement your own complex travel mode.

Example Please check the documentation of `TripRouter` (see `http://matsim.org/javadoc` → main distribution) for more details and pointers to examples.

45.2.8 Mobsim

Alternative mobsim in Java Besides adding `MobsimListener` implementations to enrich the standard mobsim, it is also possible to replace the entire mobsim by a custom implementation. A mobsim is basically a `Runnable` which is supposed to take a scenario and produce a stream of events. This allows you to use the co-evolutionary framework of MATSim while replacing the traffic model itself.

Example for alternative mobsim in Java Please check the documentation of `Mobsim` (see `http://matsim.org/javadoc` → main distribution) for more details and pointers to examples.

Alternative mobsim in another programming language Your implementation need not be written in Java. The framework includes a helper class to call an arbitrary executable which is then expected to write its event stream into a file. This pattern has been used successfully many times, see, e.g., Section 43.1, or the CUDA implementation of the mobsim by Strippgen (2009). Note that we have found consistently that an external mobsim does *not* help with computing speed: Whatever is gained in the mobsim itself is more than lost again by the necessary data transfer between MATSim and the external mobsim. As of now, we cannot yet say if newer data exchange techniques, such as Google Protocol Buffers, may change the situation. Until then, the external interface should rather be seen as the option to inject a different, possibly more realistic, mobsim into MATSim.

45.2.9 PlanStrategy

Replanning in MATSim is specified by defining a set of weighted strategies. In each iteration, each agent makes a draw from this set and executes the selected strategy. The strategy specifies how the agent changes its behavior. Most generally, it is an operation on the plan memory of an agent: It adds and/or removes plans, and it marks one of these plans as selected.

Strategies are implementations of the `PlanStrategy` interface. The two most common cases are:

- Pick one plan from memory according to a specified choice algorithm.
- Pick one plan from memory at random, copy it, mutate it in some specific aspect, add the mutated plan to the plan memory, and mark this new plan as selected.

The framework provides a helper class which can be used to implement both of these strategy templates. The helper class delegates to an implementation of `PlanSelector`, which selects a plan from memory, and to zero, one or more implementations of `PlanStrategyModule`, which mutate a copy of the selected plan.

The maximum size of the plan memory per agent is a configurable parameter of MATSim. Independent of what the selected `PlanStrategy` does, the framework will remove plans in excess of the maximum from the plan memory. The algorithm by which this is done is another implementation of `PlanSelector` and can be configured.

The four most commonly used strategies shipped with MATSim are:

- Select from the existing plans at random, which are weighted by their current score.
- Mutate a random existing plan by re-routing all trips.
- Mutate a random existing plan by randomly shifting all activity end times backwards or forwards.
- Mutate a random existing plan by changing the mode of transport and re-routing one or more trips or tours.

Routes are computed based on the traffic conditions of the previous iteration, which are measured by means of an `EventHandler`. Using the same pattern, your own `PlanStrategy` can use any data which can be computed from the mobility simulation. The source code of the standard `PlanStrategy` implementations can be used as a starting point for implementing custom behavior.

Re-routing as a building block of many replanning strategies is a complex operation by itself. It can even be recursive: For example, finding a public transport route may consist of selecting access and egress stations as sub-destination, finding a scheduled connection between them, and finding pedestrian routes between the activity locations and the stations. With the `TripRouter` interface, the framework includes high-level support for assembling complex modes of transport from building blocks provided by other modules or the core.

Please check the documentation of `PlanStrategy` (see `http://matsim.org/javadoc` → main distribution) for more details and pointers to examples.

45.2.10 Scoring

The parameters of the default MATSim scoring function (Chapter 3) are configurable. The code, which maps a stream of mobsim events to a score for each agent is placed behind a factory interface and replaceable. However, replacing it means replacing the entire utility formulation. There is currently no mechanism for composing a utility formulation from contributions by different modules. For instance, a module which simulates weather conditions would probably calculate penalties for pedestrians walking in heavy rain, and the Cadyts (Chapter 32) calibration scheme already uses utility offsets in its formulation. A modeler who wishes to compose a scoring function from the Charypar-Nagel utility, the rain penalty and the calibration offset needs to do this manually, in code, accessing the code of all three modules contributing to the score and (for instance) summing up their contributions. As of the writing of this chapter, this makes scoring in a way the least modular part of MATSim: It has to be re-defined, in code, for every combination of modules which contribute to the utility.

Keep in mind that score and replanning are not inherently coupled or automatically consistent with each other. Consider a scoring function which penalizes left-turns. This is straight-forward to program: You would iterate over every route an agent has taken. Looking at the `Network`, you would calculate for each change of links if you consider it a left-turn, and if so, add a (negative) penalty to the score. However, this would not by itself lead to a solution where routes are distributed according to this scoring function. The reason is that the default replanning only proposes fastest routes, in other words, least-cost paths with respect to travel time. By default, the plan memory of an agent will only ever contain routes which have in one iteration been a fastest route. The behavior of the router is, in this case, inconsistent with the utility formulation.

Please check the documentation of `ScoringFunction` (see `http://matsim.org/javadoc` → main distribution) for more details and pointers to examples.