Software Architecture for a Transparent and Versatile Traffic Simulation

Michael Zilske, Kai Nagel

Technische Universität Berlin
Verkehrssystemplanung und Verkehrstelematik
`michael.zilske@tu-berlin.de`

Abstract. MATSim is a traffic simulation software package which can be customized and extended in the Java programming language using a set of interfaces. Most recently, it acquired a plug-in system which was implemented using light-weight framework-assisted dependency injection, a pattern more typically used in enterprise rather than research software. We describe the interfaces and the plug-in system implementation. The architecture makes it easier and safer to combine independently developed components to complex simulation models, compared to many ad-hoc solutions often found in research software.

Keywords: traffic simulation, software architecture, dependency injection

1 Transparent and versatile simulation software

Since its inception, the MATSim project [6, also see http://matsim.org] has always aimed at being a transparent and versatile research tool. Versatility means that the tool can be used for novel research questions which are not on the mind of its inventors. Transparency means that researchers who publish results obtained with the help of the tool can and will phrase the steps which they undertook in a way which makes them easy to reproduce [3], as in "We used revision r of the software with settings S on input I, and obtained the following results." Anyone with access to the data and software will then be able to re-run the original experiment, and anyone with sufficient knowledge of the software and the domain will be able to re-evaluate the original findings.

Translated to software engineering terms, the demand for versatility requires the software to be extensible. There are certainly research problems which are novel, yet can be phrased as reductions to MATSim runs, without the need to modify the tool or indeed write any code beyond generating input and transforming output. Many applications of MATSim, however, are not of this type, but require extending the functionality of the software.

Any software product can be extended in functionality as long as the source code is available. Also, when the modified source code is in a public repository, under version control, any research citing a revision number is, in principle, reproducible. However, the further the custom version diverges from the main line, the less likely it is that another community member or a research advisor will be able to confidently interpret the results using previous knowledge of the software, because it is not clear or easily discernible which parts of the behavior of the software system have changed under global modifications.

Extending software in a transparent way means, rather than modifying code in place, to use interfaces which the software provides as hooks for extensions. In that case, the extended system consists of a previously published revision of the original software, together with custom code implementing the extension interfaces, as well as custom code wiring custom and standard components together. A full plug-in system removes the need for the last step: Extensions are only declared, not imperatively created and connected. Their instantiation is managed by the framework, which can now keep a model of which extensions are active in a particular configuration, since that concern is removed from user code.

In this paper, we discuss the architecture of MATSim as it relates to the goals of versatility and transparency. Section 2 briefly introduces MATSim just as much as is needed to explain what it means to extend it. In Section 3, its *extension points* are identified, which are interfaces against which a user can implement new functionality. Section 4 describes the implementation of the extension mechanism itself, using the lightweight Guice software framework for dependency injection. In section 5, we conclude with some remarks about possible future developments on the software side of the MATSim project.

2 Controller

MATSim is an iterative simulation. A simulation run begins with constructing an initial person-based travel demand model, which is done in user code prior to calling the simulation itself. The simulation progresses in a loop, which relaxes the demand with respect to a utility function. When a termination criterion is satisfied, the loop terminates, and control is returned to the client, where analysis code can be run.

A class called `Controller`[1] implements the simulation loop as depicted in Figure 1, starting after the initial demand generation and ending before analysis. Each iteration consists of three pre-defined process steps: `Physical simulation`, `Scoring`, and `Replanning`. At several points in the control flow, additional process steps are called which can be registered by the user as implementations of designated interfaces extending `ControllerListener`. Several implementations of a single listener type can be provided, and they are called in an undefined order. For this reason, one implementation of a listener type cannot assume a computation specified in another implementation of the same listener type to have been carried out. Similarly, the scoring and the replanning step are conceptually executed concurrently to implementations of their associated listener types.

The `ControllerListener` interface is a simple execution hook, similar to the Java `Runnable`. An instance of `ControllerListener` can expect to be called at the

[1] Wrongly spelled `Controler` in MATSim.

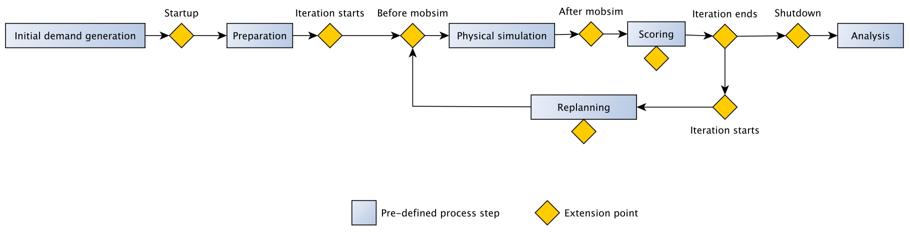

Fig. 1. The extensible MATSim simulation loop. (Adapted from [6].)

point in the control flow for which it registered, and it can expect the pre-defined process steps to behave according to contracts which can be informally described like this:

Preparation initializes the agent population read from an input file to a state where it can be executed by the physical simulation. This includes registering activity locations with network links and finding initial routes.

Physical simulation executes the travel demand of the agent population on the capacity-constrained transport network. It produces a timeline of the simulation outcome, represented by a stream of Events.

Scoring observes the Events stream, calculates for each agent a score for its outcome, and stores it in the agent memory.

Replanning lets each agent mutate its planned travel behavior.

The implementations of these process steps can be switched by the user. The standard implementations shipped with MATSim are almost completely independent components, sharing state only through two simple mutable data containers, Population and Network, and the Events stream. This makes them easily replaceable.

3 Extension Points

The simulation loop with its process steps and the ControllerListener interface have all been in place since the beginning of the MATSim project. It has also been possible to replace the implementations of the pre-defined process steps, albeit with diverging syntax. For instance, replacing the Physical simulation with a custom implementation would be possible by changing a factory property on the Controller instance, while replacing the Preparation or the Replanning stage would require subclassing the Controller class and overwriting a method.

Rather than replacing the entire Physical simulation or Replanning stage, a user will most likely want to extend or modify the behavior of their standard implementations. Historically[2], the general way of doing this has been to

[2] A previous report on the then current MATSim architecture can be found in [5]

subclass or copy the standard implementation class. However, the standard implementations of some process steps, notably Scoring and Replanning, were from the beginning conceived as *managers* of user-provided behavior: Scoring can be configured with a utility formulation, and Replanning can be extended with user-provided mutators in a genetic algorithm. Still, to practically do this, the user would at least subclass Controller and replace the *construction* of the standard implementation instance, in order to decorate it with the new behavior.

While this practice worked when each researcher was working on extending MATSim by a different aspect for one-off experiments, it fails when it comes to integrating those aspects into a single system.

Over time, we developed more and more parts of the software to be customizable through an increasingly clear-cut set of interfaces. At the same time, we removed mechanisms for ad-hoc customization unaccounted for by design, making the interfaces the designated extension points.

The rest of this section describes the standard process steps from Section 2 in sufficient detail to motivate their extension points. In Section 4, we describe the implementation of a component or plug-in system, where the user does not write and maintain component construction code, but describes components, and declares the extension points they implement, in recombinable modules.

3.1 Physical simulation and events

The Physical simulation concurrently simulates travel plans in a capacity-constrained traffic system. It is a function of a set of plans to a set of outcomes. As a software component, its dependencies are the scenario data (Population and Network), and it produces a stream of Events, describing the outcome. A custom implementation of a Physical simulation need not be written in the Java programming language: The framework includes a helper class to call an arbitrary executable which is then expected to write its event stream into a file.

The traffic flow simulation 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. Examples of such events are:

- 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 identifier, a link identifier, an activity type or other data. In theory, it should be possible to replay the traffic flow simulation 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 traffic flow simulation 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 mostly 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 spent at activities or for traveling.

MATSim extensions can observe the traffic flow simulation by interpreting the stream of Events.

3.2 Scoring

The parameters of the standard MATSim scoring function are configurable. The code which maps a stream of traffic flow simulation Events to a score for each agent is placed behind a ScoringFunctionFactory interface and replaceable. Within the factory method, users can build a custom utility formulation which can be different for each synthetic person. There are building blocks for common terms like the utility of performing an activity, or the disutility of travelling. Custom terms can be added. For instance, a module which simulates weather conditions would probably calculate penalties for pedestrians walking in heavy rain. A modeler who wishes to compose a scoring function from the standard MATSim utility and the rain penalty can re-use the former and add the latter.

3.3 Replanning

Replanning in MATSim is specified by defining a weighted set of 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, and mark it as selected.
– 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 most commonly used strategies shipped with MATSim are:

– Select from the existing plans at random, 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 of one or more trips and then re-routing them with the new mode.

Routes are computed based on the previous iteration's traffic conditions, measured by analyzing the Events stream. Using the same pattern, a custom PlanStrategy can use any data which can be computed from the physical simulation outcome.

3.4 Travel time, disutility and routing

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 service, the framework includes high-level support for assembling complex modes of transport from building blocks provided by other modules or the core.

The TripRouter provides methods to generate trips between locations, given a mode, a departure time, and an identifier of the travelling person [1]. It is used in the replanning stage of the simulation loop to modify the behavior of the simulated agents by replanning their trips. A trip is a sequence of plan elements representing a movement. It typically consists of a single leg, or of a sequence of legs with *stage activities* in between. For instance, public transport trips contain stage activities which represent changes of vehicles in public transport trips.

The behavior of the TripRouter is assembled from RoutingModule instances, one of which is associated with each mode. A RoutingModule defines the way a trip is computed, and declares the stage activities it generates.

The association between main modes and RoutingModule instances is configurable, and a user can provide custom RoutingModule implementations. This is required for use-cases which require custom routing logic, for instance if a new complex mode of transport is to be evaluated.

The standard behavior is that there are two RoutingModule implementations: For *network routing*, and for *teleportation*. By network mode, we refer to a mode whose trips are represented by paths through the network. By teleportation, we mean that travel times are determined based on a non-network property of the origin-destination relation, such as the bee-line distance. Even though no network route is calculated, this concern is implemented by a RoutingModule, because it determines the characteristics of a trip, given origin, destination and mode.

The network router, but not the teleportation router, makes use of two further interfaces: First, TravelDisutility , which answers queries about the time-dependent traversal cost of each edge, to be used by the least-cost path algorithm. And second, TravelTime, which delivers the time-dependent traversal time for each edge, which is also used by the least-cost path algorithm to advance the time as paths are expanded.

The standard implementation of TravelTime answers queries from an in-memory map from $(link, timebin)$ to $duration$, which is initialized with free-flow traversal times as determined by the road category. After each iteration, it is updated with measured travel times, which are obtained by observing the Events stream.

The standard implementation of TravelDisutility evaluates a linear function of the link length and the travel time, the coefficients of which are configuration settings.

4 Dependency Injection

We now have identified several interfaces as extension points to modify the standard behavior of the simulation. In Figure 2, we show them in a class diagram in the context of the components whose behavior they modify. Notice their different multiplicities. There is a single ScoringFunctionFactory, since it can be written to create a different function object for each agent, in case heterogeneous scoring over agents is desired. Because of the way the replanning works, there is a single PlanSelector, but there can be an arbitrary number of weighted PlanStrategy instances. There is one RoutingModule for each defined mode of transport, and there is one TravelTime and one TravelDisutility for each mode which is routed on the network.

The problem is now to provide a facility by which a user can create a Controller instance whose functionality is extended by custom implementations of one or more of these interfaces, with the additional requirement that such extensions be combineable.

At first glance, this could be done by making the Controller a JavaBean-like class with settable properties. Taking into account the different multiplicities, and the fact that some of the extension points are indexed by mode of transport, this could be usable like this:

```
Controller controller = new Controller();
controller.setScoringFunctionFactory(new RainScoringFunctionFactory());
controller.addTravelTime("bike", new BikeTravelTime());
```

Modularity could be realized by passing the Controller instance to several methods, where each method would correspond to a module, and would modify the properties of the Controller to register the extensions this module provides.

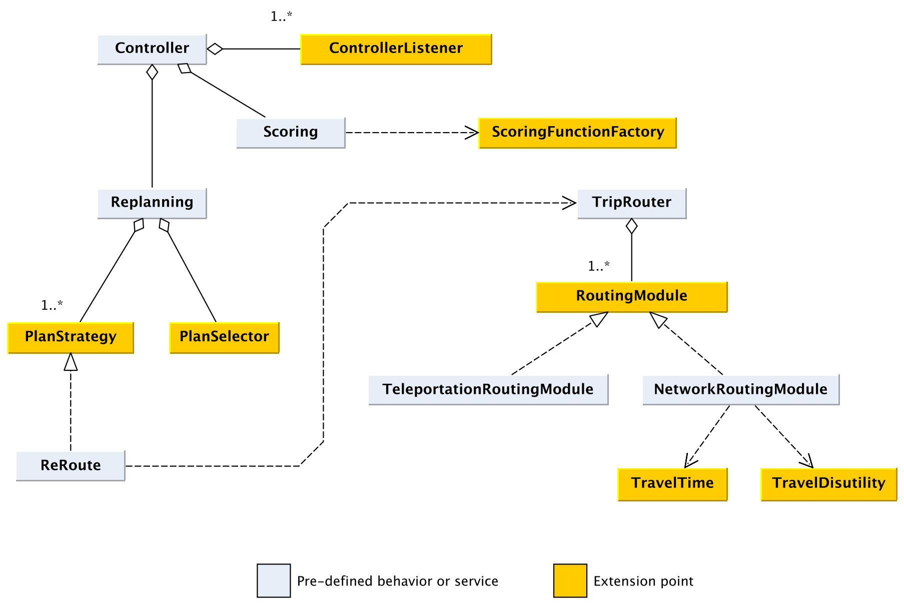

Fig. 2. Extension points in the context of the components whose behavior they modify.

The main problem with this approach is this: The no-argument constructors of the user-provided classes in the above example are an idealization. Typically, these classes will have dependencies of their own, and some of these will be services provided by MATSim. For instance, a user-defined ScoringFunctionFactory may want to use the TripRouter to judge a chosen alternative relative to others. To make this possible, TripRouter would have to be a gettable property on Controller . However, a TripRouter internally consists of RoutingModule instances, which in turn are settable properties on the same level. Hence, the TripRouter is only constructable and gettable as soon as all desired RoutingModule instances are set, and there is no easy way to enforce this, except asking the user to sort their usage of getters and setters and checking internally that no setter of A is ever called after something which depends on A has been constructed.

The fully-general, elementary solution to this problem is to require the entire object graph to be constructed bottom up, creating the most basic objects and services first and passing them to higher-level objects by their constructors. This forces the user, on a programming language level, to create the objects in the correct order, producing code like the following:

```
BikeTravelTime bikeTravelTime = new BikeTravelTime();
TripRouterImpl tripRouter = new TripRouterImpl(bikeTravelTime);
RainScoringFunctionFactory rainScoringFunctionFactory = new
    RainScoringFunctionFactory(tripRouter);
```

```
Controller controller = new Controller(rainScoringFunctionFactory,
    tripRouter);
```

With this, the property of modularity is lost. There is no general way of factoring parts of code like the above into independent methods with a common signature. Every such script by every user would have to contain the constructor calls for every required object. Moreover, this approach breaches encapsulation: While TripRouter is meant as a service interface, to be consumed by user code rather than implemented, its instantiation has now been moved to user code, exposing the implementation class.

An established solution to these problems in enterprise software is framework-assisted dependency injection [2], where dependencies within a set of managed objects are resolved by a software framework. This enables a user-side simplicity and modularity similar to the inital example, even when the user-provided classes have additional dependencies, while achieving the safety and generality of the explicit approach where all dependencies are passed by constructor.

Guice [4] is a dependency injection framework developed by Google[3]. Its core concepts are *binding* an interface type to an implementation, and *injecting* managed instances of bound types, sometimes called components, with other components. Bindings are defined within *modules*, which are Java classes. An advantage to earlier frameworks like Spring, which defined its bindings in XML files, is that module definitions are automatically included in refactoring tools provided by popular IDEs, and no additional tooling is needed. Also, since bindings are Java statements, it is easy to transform the general language provided by Guice to declare bindings into a more domain-specific language, specifically for binding extensions to our designated extension points.

An example for a module which extends two extension points by adding two bindings, one for a custom ScoringFunctionFactory, and one for a custom TravelTime:

```
bindScoringFunctionFactory().to(RainScoringFunctionFactory.class);
addTravelTimeBinding("bike").to(BikeTravelTime.class);
```

Note that these two statements do not share any variables, so they can be decomposed into two modules.

Since modules are Java code, binding statements can be made conditional upon configuration parameters. In MATSim, a data structure with user-extensible system-wide configuration parameters is read from a file and made available at startup time. At that point, a set of modules shipped with MATSim, together with user-provided modules, is evaluated by the framework to construct the object graph, which is afterwards immutable for the simulation run.

A slightly simplified version of the object graph of the standard configuration of MATSim is shown in Figure 3. Every dashed line corresponds to a binding. The dashed boxes are bound interfaces appearing on the left-hand side of bind statements, and the solid boxes are implementation types appearing on the right-hand side of bind statements.

[3] The current release version, 4.0, was used for this implementation.

Once an interface type is bound, it can be injected into instances which are managed by the framework, in particular of classes on the right-hand side of bind statements. Since such instances are created by the framework, their classes must have an injectable constructor, which is one taking either no arguments or only arguments of other bound types. When the framework creates an instance, its dependencies have already been created and can be passed to the constructor (injected).

Apart from constructors, also fields can be injected, allowing for a more concise notation. For example, the main implementation class of the Scoring stage includes an annotated field definition by which it declares a dependency to the possibly user-defined ScoringFunctionFactory component, which it then uses to create ScoringFunction instances for simulated entities:

```
class ScoringFunctionsForPopulation {
  @Inject ScoringFunctionFactory scoringFunctionFactory;
  // ...
    sf = scoringFunctionFactory.createScoringFunction(person);
}
```

The dependency is resolved by the framework at startup time. The field is injected when the instance is created. Instance methods can expect a fully initialized instance, with all dependencies resolved.

Each injection produces a solid edge in the graph.

For cases where constructor and field injection is not possible, for example when an instance needs to be built programmatically in multiple steps, a similar construct can be used to specify a custom Provider (denoted by double arrows in Figure 3), a functional interface which is called by the framework and is expected to return an instance. Orthogonally, another construct exists for binding a single concrete instance of the type or the Provider, when it is already available at configuration time (denoted by gray background in Figure 3).

Since the creation of components is managed by the framework, their dependency graph is inspectable at run-time. In contrast to the hand-drawn class diagram of Figure 2, the object diagram can be plotted and saved to the output directory of a simulation run. It conserves and visualizes the structure as determined by the set of modules and the configuration. The (slightly simplified) example of Figure 3 shows a standard configuration of MATSim, but the graph will track configuration changes and custom modules introduced by users of the framework.

5 Conclusion and Future Work

We identified and described a set of interfaces to customize and extend a traffic simulation software package. As a mechanism for the user to implement custom behavior using these interfaces, previous versions of the software relied on a system of settable factory properties on multiple levels, which had grown over time as the need for more such interfaces developed. Its main drawback was

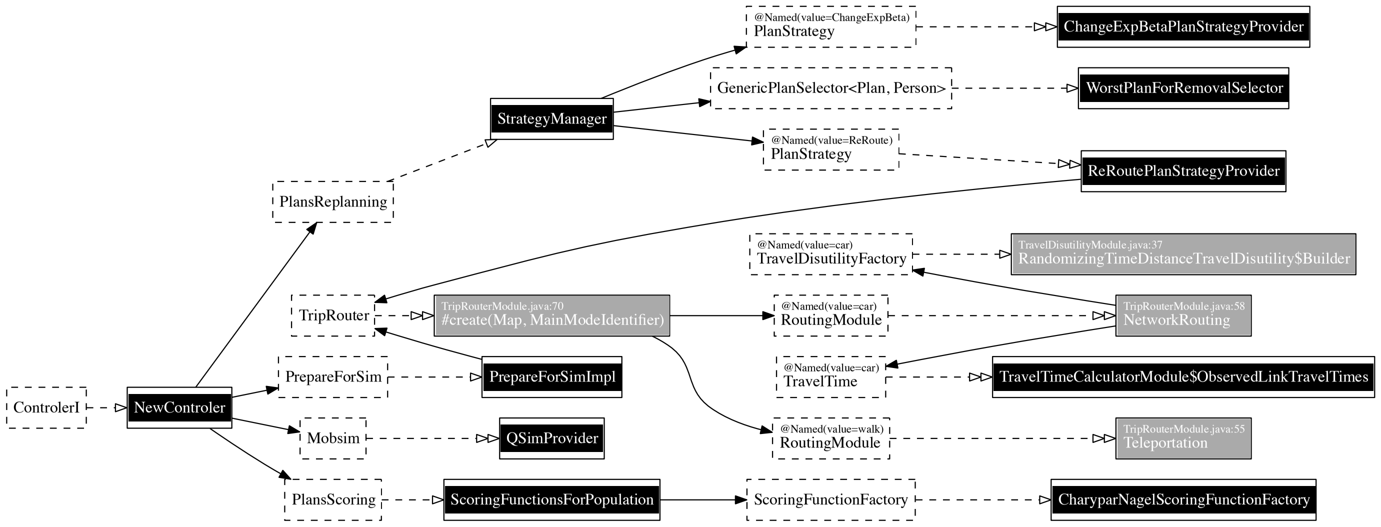
ControlerI
NewControler
PlansReplanning
StrategyManager
@Named(value=ChangeExpBeta) PlanStrategy
ChangeExpBetaPlanStrategyProvider
GenericPlanSelector<Plan, Person>
WorstPlanForRemovalSelector
@Named(value=ReRoute) PlanStrategy
ReRoutePlanStrategyProvider
@Named(value=car) TravelDisutilityFactory
TravelDisutilityModule.java:37 RandomizingTimeDistanceTravelDisutility$Builder
TripRouter
TripRouterModule.java:70 #create(Map, MainModeIdentifier)
@Named(value=car) RoutingModule
TripRouterModule.java:58 NetworkRouting
PrepareForSim
PrepareForSimImpl
@Named(value=car) TravelTime
TravelTimeCalculatorModule$ObservedLinkTravelTimes
Mobsim
QSimProvider
@Named(value=walk) RoutingModule
TripRouterModule.java:55 Teleportation
PlansScoring
ScoringFunctionsForPopulation
ScoringFunctionFactory
CharyparNagelScoringFunctionFactory

Fig. 3. The run-time component graph of a typical MATSim instance. The graph is created at startup time, determined by a set of system and user-defined modules and their configuration. It is immutable for the duration of a simulation run. Since components are framework-managed, the graph can be plotted and saved automatically with the simulation output. This is the graph resulting from the MATSim standard configuration. It was manually simplified to fit the page and to remove experimental extension points not mentioned in the text, as well as trivial or technical components such as proxies.

that the responsibility of the correct sequence of object creation was left to the users, who had to find out by themselves at which point in the control flow all dependencies necessary to create an instance of a custom class would be available, while the instance itself would not yet have been requested. If such a point did not exist, the creation of the entire object graph would have to be done in user code, which would introduce duplicate code over different configurations, and expose all dependencies of all components, requiring all configuration code to change whenever a constructor signature changes.

For this paper, that mechanism was developed into a plug-in system. It was implemented using dependency injection provided by the Guice framework. This solved the problem of mutable factory properties, unspecified order of object creation, and public visibility of dependencies. All components are constructed at startup time, in an order induced by the dependency graph, and with a check on completeness and uniqueness. The construction process and the resulting component graph are inspectable. This can be used to draw configuration diagrams and save them with the simulation output, so that the precise configuration used to produce a simulation run is transparent.

Framework-assisted dependency injection is a pattern more typically used in enterprise rather than research software. We are tracking other developments in the Java world which may be put to use for system-level simulation development. There is a pattern called reactive programming (see e.g. [8]), which combines syntax similar to the Java 8 Stream interface with the Observer pattern. This could be put to use in the Event architecture. It provides concise constructs for subscribing, unsubscribung, filtering, merging, grouping and flattening asynchronous streams. This pattern may be of interest for the MATSim event processing mechanism.

The process step summarized under physical simulation is a complex, extensible piece of software by itself. A close-up view would identify another set of extension points on that layer. This is also where interaction between agents takes place. MATSim does not use any framework for agent-based microsimulations, but implements all concepts, including concurrency, directly in Java. One of them are coroutines and light-weight threads, which may in principle allow every simulated entity to run in its own execution thread. In contrast to reactive programming or other styles which invert control flow, code can be written like a script for the simulated entity. This is an advantage even if parallelism is not taken into account. The concept has most recently been re-popularized by the Go programming language, but Java does not have an equivalent construct. Project Quasar [7] implements it on the bytecode instrumentation level.

Bibliography

[1] Dubernet, T.: The new matsim routing infrastructure (2013), presentation at the MATSim User Meeting, Zurich

[2] Fowler, M.: Inversion of control containers and the dependency injection pattern (2004)

[3] Gandrud, C.: Reproducible Research with R and RStudio. CRC Press (2015)

[4] Google, Inc.: Guice, https://github.com/google/guice

[5] Grether, D., Nagel, K.: Extensible Software Design of a Multi-Agent Transport Simulation. Procedia Computer Science 19, 380–388 (2013)

[6] Horni, A., Nagel, K., Axhausen, K.W. (eds.): The Multi-Agent Transport Simulation MATSim. Ubiquity Press (in preparation), see http://ci.matsim.org:8080/view/All/job/MATSim-Book/ws/main.pdf

[7] Parallel Universe: Quasar, http://docs.paralleluniverse.co/quasar/

[8] Project Reactor: Reactor, http://projectreactor.io/