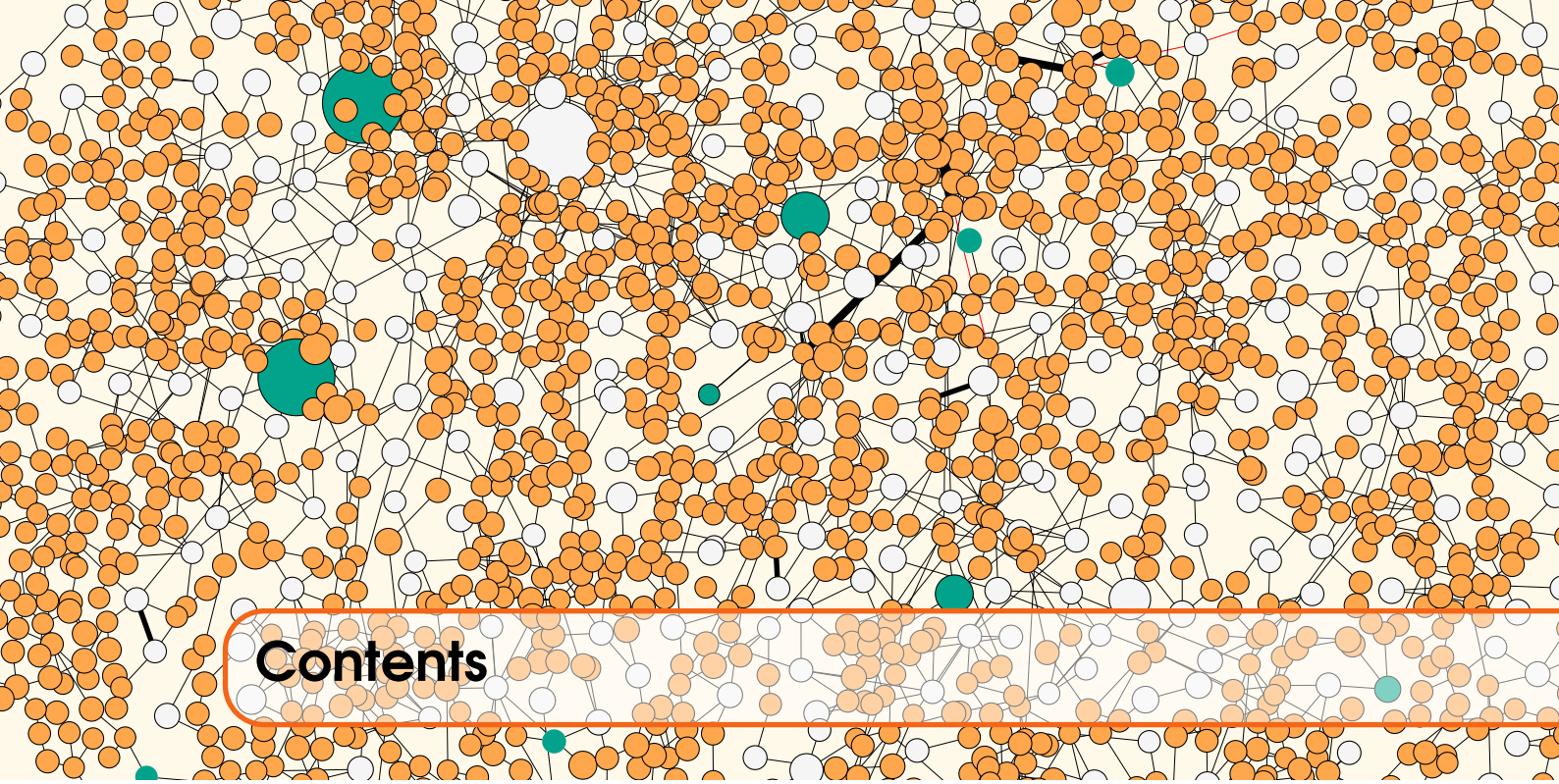
# SFINA Manual

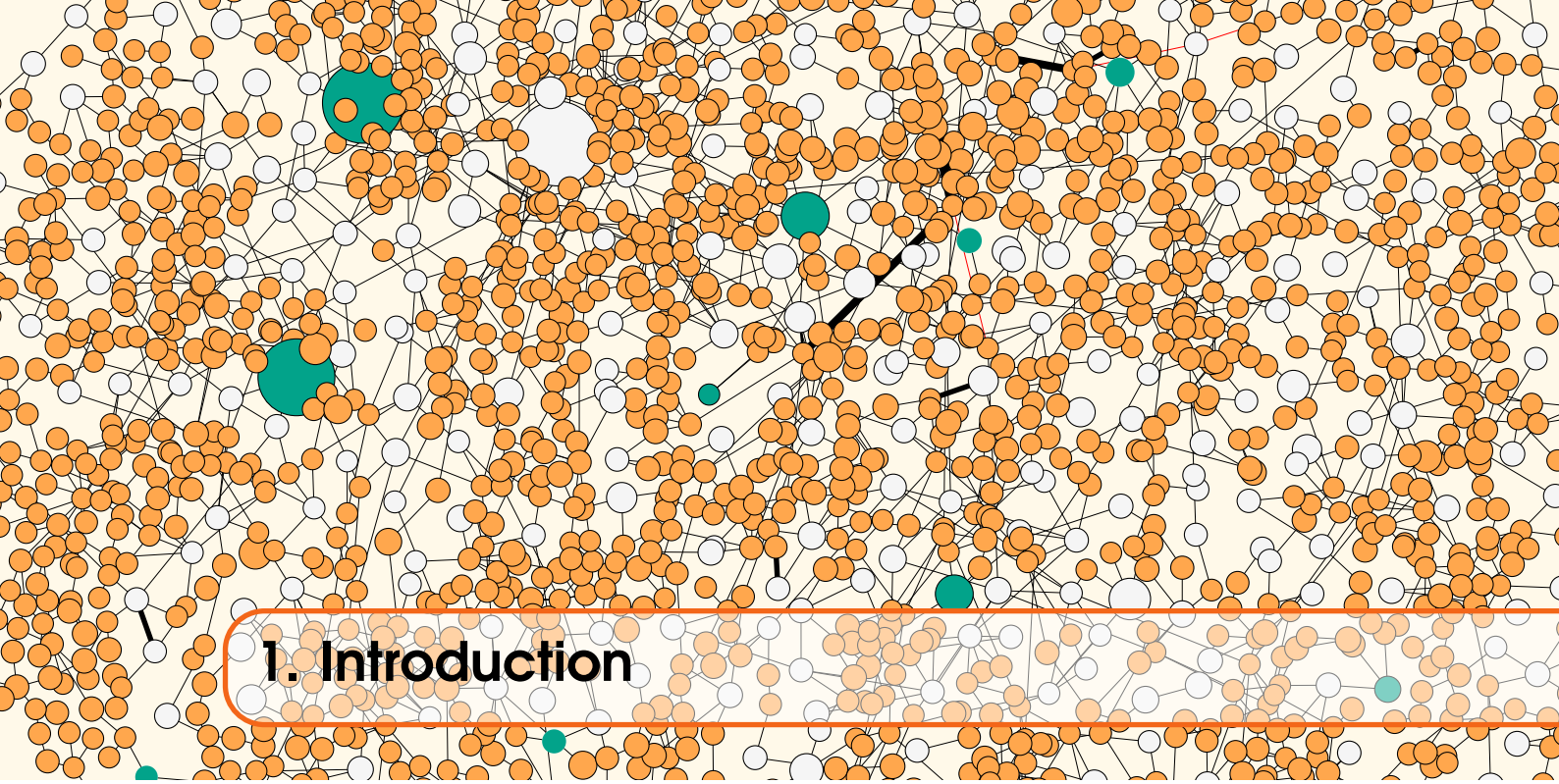
## Simulation Framework for Intelligent Network Adaptations

Chair of Computational Social Science, ETH Zurich



# Contents

# General



# 1. Introduction

This manual explains the functionality of SFINA and gives hands-on advice on how to modify and extend it for one's own needs. If you implement new core functionality, we would be happy to integrate it in the official release for everyone to use. So please contact us in this case or if you have feedback for improvement.
SFINA is a simulation tool for flow networks, highly adaptable for different application domains and scenarios. A flow network is a collection of nodes between which quantities can be exchanged through links. For example in power networks, the nodes are generator/load buses connected by transmission lines through which real and reactive power can flow. Currently a fully functional power simulation is implemented, including cascading failure simulation under different attack scenarios. We are also working on extending the functionality to simulate multi-layer inter-dependent networks. For the future we envision the implementation of more domains such as information, transportation, water or gas. This will provide a unique tool to simulate several scenarios of interconnected multi-domain networks.
This simulation tool is intended to be used for researchers to study the stability, growth and adaptation of flow networks. The goal is to support decision and policy-making by providing a tool for optimization. However, by keeping it very flexible and adaptable, we hope that many others will use it for their own studies with different approaches and use-cases.

## 1.1 Install and setup

Depending on the goal one may want to get the full source code of SFINA or only the compiled files for easy use of the current functionality.

### 1.1.1 Using the compiled code

This is the way to go, if the current core functionality is sufficient, meaning the currently implemented domain and backends for this domain. It is still possible to implement complex functionality for different experiments on top of this.

1. Download SFINA.jar and necessary libraries from the website.
2. Create your own project in the IDE of your choice (Eclipse, Netbeans or others).

3. Import SFINA.jar as an external library.
4. Create an experiment by following the instructions in section 1.2 on how to run simulations of already implemented applications or in section 3.1 on how to create your own application.

### 1.1.2 Using the full source code

If you want to extend the core functionality to your needs, namely adding new domains or new backends for existing domains, then follow these steps:

1. Get the sources by cloning from Github into the current folder on your computer. For this open a terminal window, navigate to the folder you want to use and type:

```
git clone https://github.com/epournaras/SFINA.git
```

2. Import the project into an IDE (SFINA is developed on Netbeans, so using this would be the easiest to setup).
3. Add the necessary packages, which can be found in SFINA library folder.
4. Start to adjust it to your needs by following the instructions in the following chapters.

## 1.2 Performing simulations and measurements

If the application one wants to use is already implemented and the goal is to only execute it with own parameters and input data, this section explains how to do so. In this case the algorithm and the types of performed measurements cannot be changed. But still several things can be adjusted in order to run different experiments: The name of the experiment, for how many time steps to run the simulation, the backend to use, any event to be executed at any time and when to reload a network from files or when not to do so. In order to run a simulation, one should follow these steps:

- Create a folder in experiments for this simulation, and put an input folder. Let's call our simulation "test", then we would need to create experiments/experiment-test/input
- Put all the necessary input files there
  - sfinaParameters.txt
  - events.txt
  - backendParameters.txt
  - time_1/topology/nodes.txt and links.txt
  - time_1/flow/nodes.txt and links.txt
  - More time folders in the same structure as time_1, if we want to restore the network at some time steps to its initial state, or if we want to load a different configuration. For more details on the input files and how to format them, see section 2.1.2.
- Specify which domain and backend to use in sfinaParameters.txt and any other backend specific necessary parameters in the backend parameter files
- Optionally put events to be executed in events.txt. Here it is also possible to define the reloading of the first input folder by putting at time n, where this should happen:

```
n,system,-,-,reload,1
```

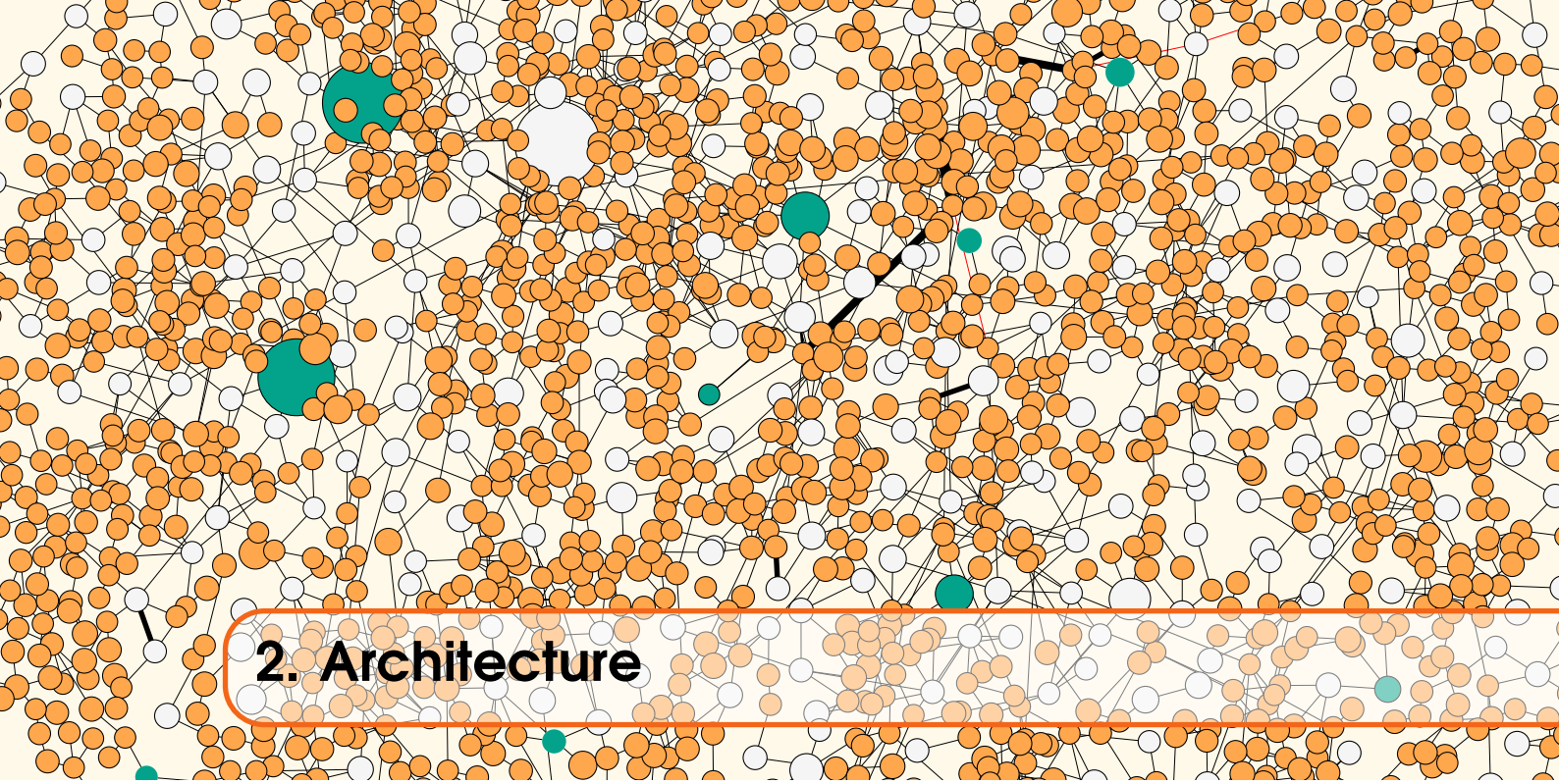
- Create a new class (e.g. testExperiment.java to run the PowerCascadeAgent.java application) in the following way:

```
import application.PowerCascadeAgent;
import applications.BenchmarkLogReplayer;
import protopeer.Experiment;
import protopeer.Peer;
import protopeer.PeerFactory;
import protopeer.SimulatedExperiment;
import protopeer.util.quantities.Time;
```

```
public class testExperiment extends SimulatedExperiment{

    // define a name for this experiment
    private final static String expName="test";
    private static String experimentID="experiment-"+expName;

    // time steps including bootstrap time
    private final static int runDuration=10;

    // other simulation Parameters
    private final static int bootstrapTime=2000;
    private final static int runTime=1000;
    private final static int N=1;

    public static void main(String[] args) {
        Experiment.initEnvironment();
        testExperiment test = new testExperiment();
        test.init();

        // create the instance of PowerCascadeAgent,
        // contained in a Protopeer peer
        PeerFactory peerFactory=new PeerFactory() {
          public Peer createPeer(int peerIndex, Experiment experiment){
              Peer newPeer = new Peer(peerIndex);
              newPeer.addPeerlet(new PowerCascadeAgent(
                      experimentID,
                      Time.inMilliseconds(bootstrapTime),
                      Time.inMilliseconds(runTime)));
              return newPeer;
            }
        };
        test.initPeers(0,N,peerFactory);
        test.startPeers(0,N);

        // run the simulation
        test.runSimulation(Time.inSeconds(runDuration));

        // analyze measurements
        BenchmarkLogReplayer replayer =
        new BenchmarkLogReplayer(expName, 0, 1000);
}
```

First notice, that our testExperiment class extends SimulatedExperiment, which is a Protopeer class (section 2.5) providing useful functionality like time and measurements. Then it is necessary to assign to expName the same name as the input directory is called, in order for the experiment to find the files. The runDuration variable defines for how many time steps the simulation will run, including the bootstrap time, which is used to initialize the experiment. Defining the bootstrapTime as 2000 ms and the runTime as 1000 ms, a runDuration of 10 corresponds to 8 simulation steps. So in general: *runDuration = bootstrapTime/runTime + number of actual simulation steps*.
The only thing left to do, is to initialize a peer which creates the PowerCascadeAgent, and finally executing it with test.runSimulation(...).
All the measurement results are saved to a binary file in peerlets-log/experiment-test/peer-0, from where it can be loaded after the simulation finished to compute, display and output measurement results. This is done automatically when initializing the provided BenchmarkLogReplayer object,

like shown in the last line of the above example.
To see what is going on during the simulation, logging is useful. For this first it is necessary to make sure in conf/log4j.properties the line “log4j.rootLogger=info, I, stdout” isn’t commented out. This will show information during the simulation and also the measurement output in the console and in the file log/info.log. To get a more fine grained output, uncomment “log4j.rootLogger=debug, D, stdout” in the same .properties file.
Now you can go ahead and run this class.



# 2. Architecture

For running experiments which are already implemented it is only necessary to understand the file system and the concept of events. For a deeper understanding and to implement own applications, the information about the flow network, simulation agent and Protopeer will come in handy.

## 2.1 File System

The SFINA file system allows easy input and output of all the network information needed for the simulation. Every agent has its own file system, where for each experiment a new directory has to be created and provided with input data and configuration files. The simulation automatically performs the output in the same format as the input, i.e. one folder per time step, however extended by finer grained simulation steps (iterations). The notion of time and iterations is explained in the following paragraphs. The file system is illustrated by figure 2.1, where input files or folders that show a solid contour have to be provided, the dashed ones are optional, as explained in more detail below.

### 2.1.1 Location of the files

Each experiment is assigned a string identifier, defined when running experiments (section 1.2). The input files of an experiment with the ID “someExperiment” have to placed in the folder experiments/experiment-someExperiment, where “someExperiment” can be any string.

### 2.1.2 Configuration files

Three configuration files are part of an experiment, providing general settings for the experiment like the domain and backend to use (sfinaParameters.txt), settings for the currently used backend (backendParameters.txt) and defining events to be automatically executed (events.txt). The first has to be provided, not doing so will result in an error. The backend parameter file doesn’t have to be provided (i.e. will not result in an error), however depending on the backend certain parameters might be necessary. This is for example the case of AC or DC in the power domain. The event file can be provided, but the simulation will also run without it.

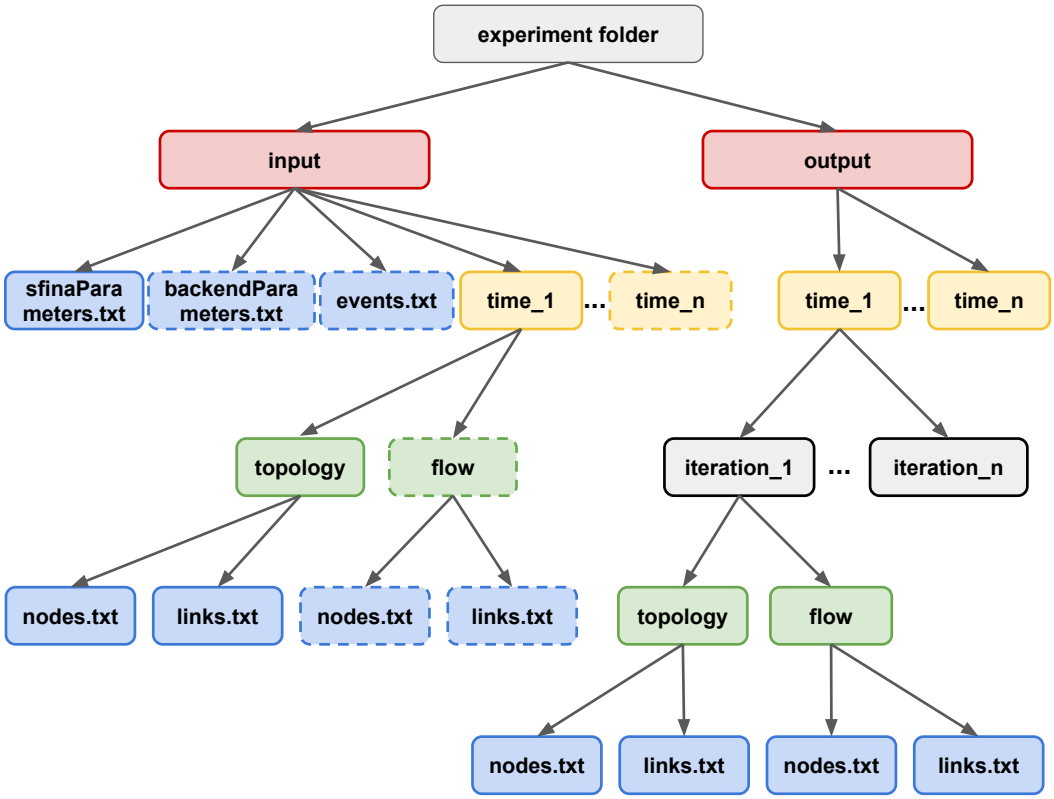



Figure 2.1: Visualisation of the SFINA file system. In input, files/folders with dashed boxes don't have to be provided. The output folder is created automatically.

**The SFINA and backend parameters files** are specified in the format "name=value". The currently available settings are summarized in the following table.

| File | Name | Possible values |
|---|---|---|
| sfinaParameters.txt | domain | power |
| | backend | matpower,interpss |
| backendParameters.txt (power domain) | flowType | AC, DC |
| | toleranceParameter | value of type double |

Table 2.1: Parameters that can be specified in the parameter files.

**The events file** has the following format, defining one event per line with comma separated values:

```
time,feature,component,id,parameter,value
2,topology,link,10,status,0
3,flow,link,9,resistance,0.1
3,system,-,-,backend,interpss
...
```

For a more detailed explanation how events events work and which ones are available, see section 2.2. The events are loaded at the beginning, but each of them is executed at the time specified in the first column. It allows to define certain important events before running the experiment and without the need to modify the code.

### 2.1.3 Network data

Each time_n folder in input as well as in outputiteration_m contains the full network data, separated into general topological information and domain specific flow information. All the output files are generated automatically, so only the input folder has to be provided in order to run an experiment. A folder time_n in input is loaded at time step n and replaces all the network information (both topological and flow) from earlier times. At least the time_1 folder with data has to be provided in order for the experiment to have sufficient data to run properly. If a folder at any time step is present, it will automatically be loaded, discarding the data from before and replacing it with the new one. On the other hand it will continue with the information from the time step before, if no folder for the current time is present.

#### Topological data

The topological data provides the nodes/links with a unique ID, which can be any string, and a status attribute which specifies if it is active or not. In the latter case the simulation treats it as if it was not present, but it can be activated during runtime for example by events. Links are directed, which can be specified by the ids of the nodes where it starts and ends. An undirected link can be simulated by creating two links, one in each direction. The formatting of the files is shown in figure 2.2.

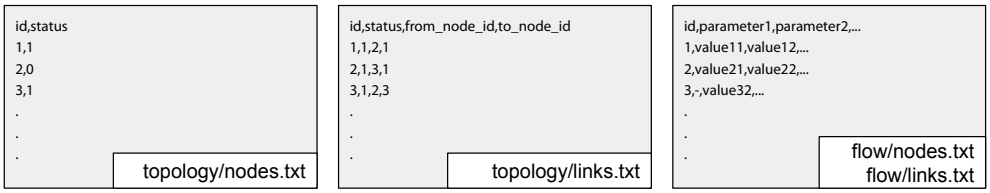



Figure 2.2: Format of topology and flow input files.

#### Flow data

The flow data is very flexible and has the same format for both nodes and links, as shown in figure 2.2. For each node/link information can be loaded which is necessary for the current domains flow simulation. For example in the power domain this might be the generation output or load of a node or the resistance of a link. The values are then stored in the nodes and links of the ID specified in the first column.

The flow data is domain specific and therefore has to be defined seperately for each one. Currently this is the case for the power domain, other domains are not supported yet.

In the most general case of purely topological networks, the flow data could be used to assign weights to the links or nodes. However in general it is not necessary to provide flow input files, in that case a topological network with no further attributes is generated.

Some nodes or links might be of a different kind than others, making it necessary to assign certain values to them but not to the others. In the power domain this is for example the case for nodes that generate power, which additionally need among others the power generation output information. In this case a dash (-) can be used to exclude this information from all other nodes as seen for value31 in the example above, it will be ignored during loading.

### 2.1.4 Manually loading network data

As explained in section 2.4 about the simulation agent, at the beginning of each time step, the corresponding input data is automatically loaded, if provided by the user. It is however also possible

to initiate data loading manually, by calling the method loadData(time), where “time” is a string to be replaced with the time matching the input folder time to be loaded. It is also possible to trigger this “reload” of the time x input at time y by an event, as explained further below.

### 2.1.5 Changing the file system structure

The structure of the file system and names of folders outlined above is defined in a configuration file, which is loaded at bootstraping of the simulation agent (section 2.4. It is placed in the folder conffileSystem.conf. Besides the names of all the folders and parameter files, it also defines the column separator and the string which is used for excluding data for some of the nodeslinks (missingValue) in the input data files, as explained above. These values can be changed, which is however not recommended.

## 2.2 Events

Events are designed for making changes to the simulation or to the network structure during runtime. They can be used in two ways:

1. Specified in events.txt to be loaded at the beginning and executed automatically at their specified time.
2. Written in the code of applications in order to change parameters “online”.

An event is defined by six parameters, specifying what action is to be executed at which time step. An overview of these parameters is given with the following table:

| Parameter | Value/Description | | | | |
|---|---|---|---|---|---|
| **time** | When the event is executed | | | | |
| **feature** | topology | | flow | | system |
| **component** | node | link | node | link | n/a (placeholder ’-’) |
| **id** | id of the link/node whose information is changed | | | | n/a (placeholder ’-’) |
| **parameter** | status | - status<br>- start node id<br>- end node id | Any flow information which was loaded or added | | - domain<br>- backend<br>- reload |
| **value** | 0,1 | - 0,1<br>- new node id<br>- new node id | New value | | - new domain<br>- new backend<br>- Time from which input data should be reloaded |

Table 2.2: Summary of currently available events and how to define them.

### 2.2.1 Using events.txt

Each line in the events configuration file defines one event, with comma separated entries for each of the six categories introduced above. As a general rule, the entries should correspond to the same strings which are also used in the other files, namely sfinaParameters.txt, backendParameters.txt and the data files. For example in the power domain if the goal is to change the resistance of a specific link with id 9 at time 3, it would be:

```
time,feature,component,id,parameter,value
3,flow,link,9,resistance,0.1
```

In the case of a system parameter change, component and ID are not applicable. In this case, just put a dash (-) instead. For example to change from power to gas domain at time 10:

```
time,feature,component,id,parameter,value
10,system,-,-,domain,gas
```

See section 2.1 about the file system for more details.

### 2.2.2 Using events in an application

The same categories are used to define events in the code, however instead of strings the appropriate Enum types are used. The constructor of the Event class takes as arguments:

```
Event(int time,
        EventType eventType,
        NetworkComponent networkComponent,
        String componentID,
        Enum parameter,
        Object value)
```

Let's look at an example, which deactivates link 20 at time step 10:

```
Event event = new Event(
        10,
        EventType.TOPOLOGY,
        NetworkComponent.LINK,
        "20",
        LinkState.STATUS,
        false);
getEvents().add(event);
```

## 2.3 Flow Network

The flow network is an object containing all the nodes and links and providing several useful methods, explained in the following.

### 2.3.1 Activation status and connectivity

First of all the flow network takes care of adding nodes/links to and removing them from the network and updating the network topology accordingly. The two basic topological properties a node/link can have are

1. isConnected(): A node which has (activated) links attached, or a link that has both (activated) start and end node, is connected.
2. isActivated(): A node/link that is functional, is activated. Deactivating a node, will not deactivate the attached links, but just disconnect them. Likewise, a disconnected node is still activated.

Once the network is loaded, it is possible to activate/deactivate nodes/links in the network, which will include/exclude them from any computation but will not remove them entirely. This way they can be used again later on. The nodes and links also keep track of their connectivitiy, i.e. if any other objects are connected or not. This information can be retrieved by the method isConnected(). An important method in the network is computeIslands() which extracts disconnected components from the topology of the network and returns them as an ArrayList containing a new flow network for each island. It takes all activated nodes/links into account, as can be seen in figure 2.3.

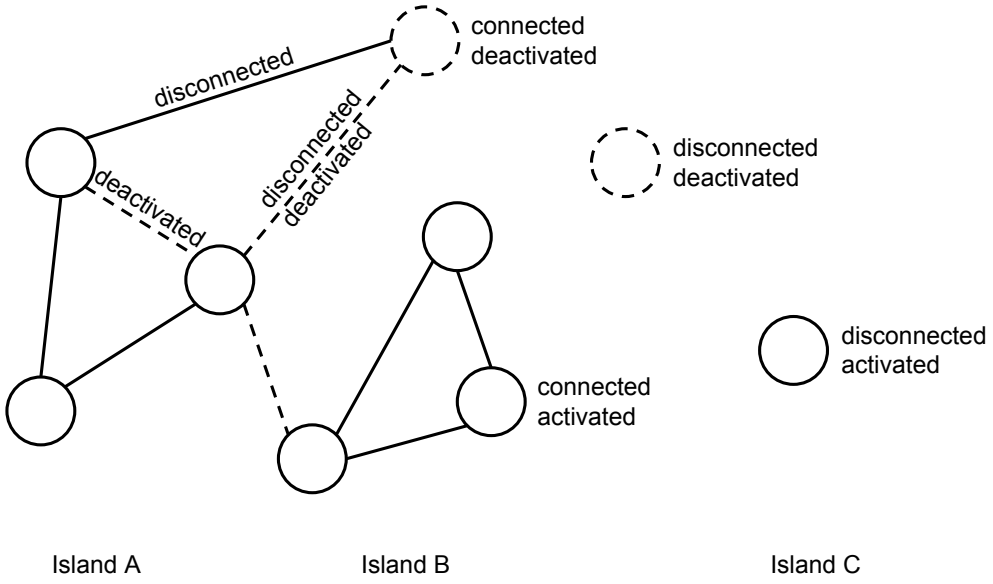



Figure 2.3: Illustration of the connectivity and activation status of nodes and links, and how they affect the computation of disconnected compontents (i.e. islands).

### 2.3.2 Metrics

Furthermore the flow network provides some useful methods to calculate metrics like degree distribution, clustering coefficient or to find the shortest path between two nodes. Some of these methods use the openly available JGraphT library. A more detailed list can be found in the method summary in section 5.2.

### 2.3.3 Flow and capacity

The flow network provides a flexible way to define the flow and capacity, meaning which physical quantities (contained in the nodes and links) is the measure of flow or how much flow a node/link can support. Use setLinkFlowType(...) and setLinkCapacityType(...) and for the nodes respectively for this purpose. This is done in the setFlowParameters() method in the simulation agent, where each domain has its default settings. When developing a new application, the developer is expected to check these. Default values are summarized in the following table.

| Domain | Link flow type | Node flow type | Link capacity type | Node capacity type |
|---|---|---|---|---|
| Power | Real power flow from | Voltage magnitude | C rating | Max. voltage |
| ... | | | | |

Table 2.3: Default values for capacity and flow for the different domains.

### 2.3.4 Accessing the flow network and its values

Some hints which might come in handy when writing applications:

1. When writing an application which extends the SimulationAgent, the current flow network can always be retrieved by the method getFlowNetwork().

2. To add new flow information to a node or link, use the .addProperty(...) method.
3. To get or change the flow information already contained in the nodes or links, use the .getProperty(...) or .replacePropertyElement(...) method respectively. The former requires casting the return value to the correct type.

Here you see these in action, doubling the real power demand of node 15 in the current flow network:

```
FlowNetwork net = getFlowNetwork();
Node someNode = net.getNode(15); // node having ID 15
double pwr =
        (Double)someNode.getProperty(PowerNodeState.POWER_DEMAND_REAL);
someNode.replacePropertyElement(PowerNodeState.POWER_DEMAND_REAL, pwr*2);
```

A summary of methods can be found in the appendix (section 5.2).

## 2.4 Simulation Agent

At the heart of any simulation is the simulation agent. Applications extend it and can therefore make use of its functionality (see section on adding applications for more detail). The simulation agent takes care of performing several tasks which are necessary for any simulation. These include keeping track of simulation steps, loading and performing output, executing events and calling key methods at every time step. The time evolution of the simulation agent is illustrated by figure 2.4.

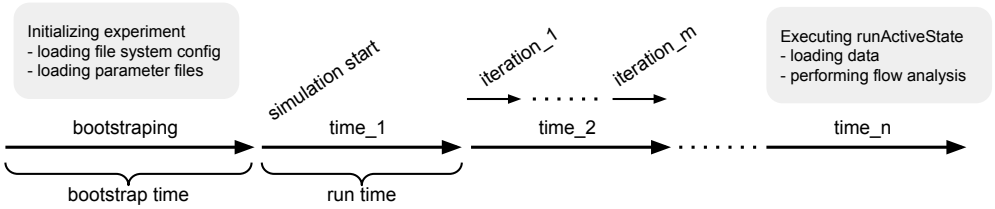



Figure 2.4: Time steps and iterations performed by the active state.

### 2.4.1 The Active State

The runActiveState() method is the main runtime of the simulation agent (**compare to passive state?**). After bootstraping, during which the experiment is initialized by loading the configuration files from the experiment folder, the main simulation is orchestrated by the this method. It is executed at every time step automatically and initiates the above mentioned tasks, as depicted in detail in figure 2.5.

The SimulationAgent class is providing this minimal but extensible structure, without implementing any specific algorithm on how to perform the flow analysis at each time step. Such algorithms are meant to be implemented in applications by overriding

1. initialStateOperations()
2. runFlowAnalysis()
3. finalStateOperations().

In the SimulationAgent, the initialStateOperations and finalStateOperations methods are empty, whereas in runFlowAnalysis() two other core methods are called, namely callBackend(), to calculate the actual flow distribution in the network, and nextIteration(), to make a data output. These two methods can be called as often as necessary, for example if a simulation at the current time step requires several iterations. It is up to the developer of an application to call the backend and to

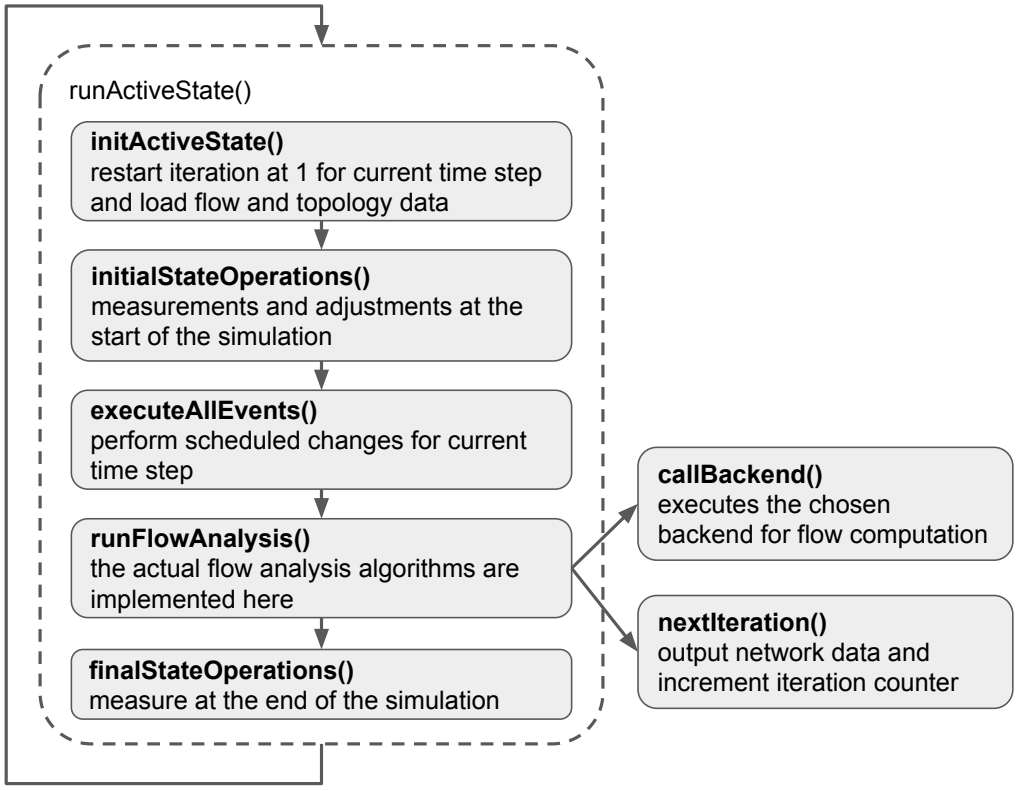



Figure 2.5: Methods executed by the active state in each time step.

initiate the output when appropriate (see section 3.1 on implementing new applications for more detail).
The two methods initialStateOperations() and finalStateOperations() are especially useful for making measurements before and after the flow analysis. Also this allows to implement measurements, which depend on the state before and after the flow analysis. However, they can also be used to implement additional functionality, especially adjustments to the network at the beginning through events, which are then executed automatically in the executeAllEvents() method (more details in section 1.2 about measurements).
The callBackend() method executes the backend, which is automatically selected according to the parameters set in the parameter files. Currently implemented backends are shown in the followig table.

| Domain | Backend | String to use in sfinaParameters.txt |
|---|---|---|
| power | interPSS | interpss |
| | Matpower (Matlab Required) | matpower |
| ... | ... | |

Table 2.4: Currently integrated backends to calculate flow in a network.

### 2.4.2 Iterations and output

The core simulation agent (SimulationAgent) provides the important method nextIteration() which calls the output function at the current iteration and then increments the iteration number. However, it is not called automatically, but it is up the developer, who implements an application (i.e.

overwriting runFlowAnalysis()), to call it when appropriate. Every time it is executed, a new output in the current time step but in a new subfolder corresponding to the current iteration is created.

## 2.5 Protopeer Toolkit (redo, maybe split in part here and appendix)

SFINA makes use of the Protopeer Toolkit which was developed at EPFL university Lausanne to provide a framework for peer to peer distributed experiments (paper). It provides useful basic functionality like time based events and measurement methods. Another important concept is peers and peerlets, which allow a fully decentralized deployment. A peer is an independent simulation instance, which can communicate over the network with other peers. Every peer can contain several peerlets, which allow to further split up a peer in seperate simulations. Currently experiments conducted with SFINA initialize just one peer with one peerlet. However this potentially allows for a very flexible use of the SFINA framework in the future.

# II For developers

# 3. New applications

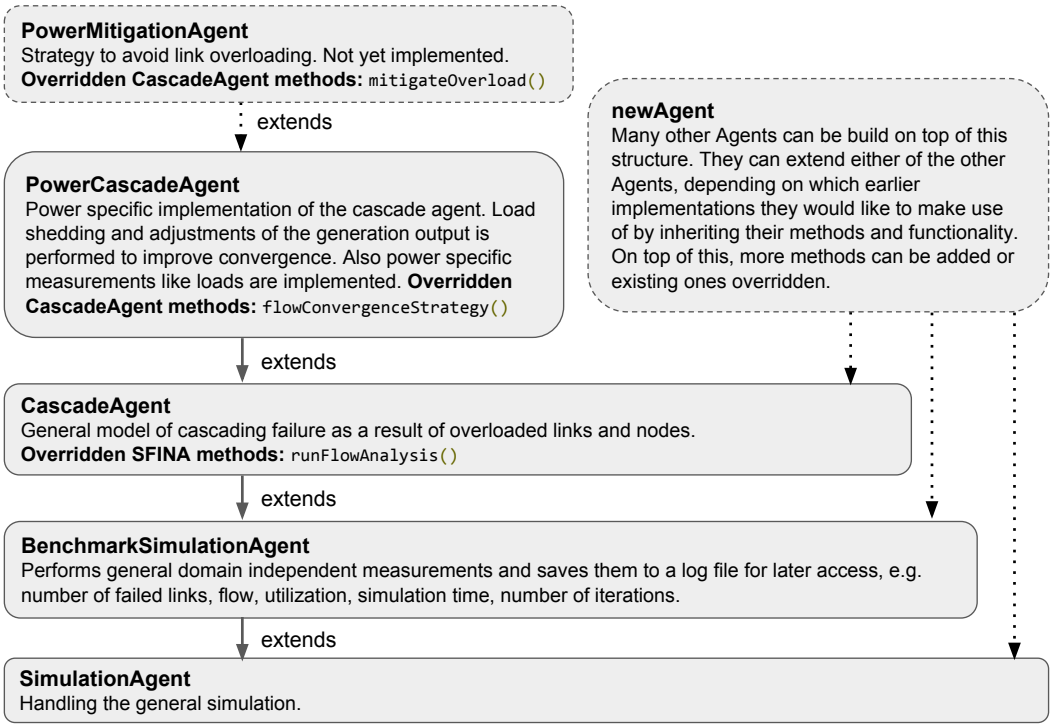



Figure 3.1: Architecture of the applications. The ones with dashed solid contours are already implemented, the dashed-contour applications are examples for possible extensions.

## 3.1 Application architecture

SFINA is designed such that it is flexible and can be easily extended. These extensions can be the implementation of a new application as described in this section, more sophisticated measurements, and more. Currently implemented applications for showcasing the capabilities of the framework as well as for providing basic functionality, are an agent for benchmarking (i.e. making measurements), a model for cascading failures and a specific implementation of the latter for power grid simulations. New applications can be build on top of them in order to make use of their functionalities. If this is not of interest to the developer, totally new applications can be implemented on top of the SFINA core functionality. In any case, whether the SimulationAgent or one of the existing applications are used for the basis of new applications, the capability of Java to extend classes and override methods is used, which will be explained in more detail below. In figure 3.1 the current applications are shown, as well as how possible extensions could be realized.

## 3.2 Implementing an Agent

Let's have a look on how extending one of the existing agent works, taking the example of the Benchmark Agent. Its beginning looks as follows:

```
public class BenchmarkSimulationAgent extends SimulationAgent{

    private static final Logger logger =
        Logger.getLogger(BenchmarkSimulationAgent.class);

    private HashMap<Integer,
        HashMap<String,HashMap<Metrics,Object>>> temporalLinkMetrics;
    private HashMap<Integer,
        HashMap<String,HashMap<Metrics,Object>>> temporalNodeMetrics;
    private HashMap<Integer,
        HashMap<Metrics,Object>> temporalSystemMetrics;
    private long simulationStartTime;

    public BenchmarkSimulationAgent(String experimentID,
            Time bootstrapTime,
            Time runTime){
        super(experimentID,
                bootstrapTime,
                runTime);
        this.temporalLinkMetrics=new HashMap();
        this.temporalNodeMetrics=new HashMap();
        this.temporalSystemMetrics=new HashMap();
    }
    // Methods
}
```

The first thing to notice is the first line, where we define that the BenchmarkSimulationAgent extends the SimulationAgent. This makes all public methods of the latter available for our Benchmark Agent. Then some variables are defined which are needed for this Agent to store measurement values during runtime. The constructor has to take at least the variables which the SimulationAgent's constructor needs and passes them on by using the super() method. More variables that are just used for this agent can be added here as well.
Next let's look at the methods in this agent. Most of them are just new methods necessary for the measurements, for example for initializing the measurement variables in each epoch. The important methods however are the overwritten ones, for example:

```
@Override
public void performInitialStateOperations(){
        // some methods
}

@Override
public void performFinalStateOperations(){
        // some methods
}
```

They are defined in the SFINA Agent but don't have any meaningful implementation there. By overwriting them here, we can "plug in" our own functionality, in this case making measurements before and after the actual flow simulation (see the next section on how to implement measurements for more details).
The SFINA agent is the core of every simulation, and is not supposed to be changed except if the implementation of a totally new domain or backend is needed (see sections 4.1 and 4.2 of this tutorial). Like the methods just introduced, it provides a general structure with methods that can be overwritten, namely:

1. public void performInitialStateOperations()
2. public void runFlowAnalysis()
3. public void performFinalStateOperations()
4. public void scheduleMeasurements()

To be precise, these are the only methods of the SimulationAgent that should be overwritten when implementing applications. If your specific application doesn't seem to fit in this structure, then you maybe didn't try enough to simplify it, or the current implementation of the SFINA Agent is not as general as it can be. If this is the case, then you're welcome to adjust it to your needs and we would be happy to hear about your suggestions and incorporate the changes ourselves.

### 3.2.1 Example: PowerCascadeAgent

This is an example of a more sophisticated application. As can be seen in figure 3.1 there are two levels:

1. Cascade Agent: Implementing a general cascade algorithm with a method that checks the links for overloads (i.e. the flow in the link exceeding its capacity). It implements (overrides) the runFlowAnalysis() method of the SimulationAgent, but also introduces a new method, namely flowConvergenceStrategy() which is an intermediary step before calling the backend to calculate the power flow in the network. This allows a very flexible implementation of different (domain dependent) necessary adjustments. This is exactly what is implemented by the Power Cascade Agent, which brings us to the next point.
2. The Power Cascade Agent mainly implements flowConvergenceStrategy() from the Cascade Agent. It first checks if the current island just consists of one isolated node, returning directly non-convergence (i.e. blackout) in this case. Then it checks if there is a generator present in the island, because without power supply the island is also blacked out. Furthermore it tries to improve convergence, i.e. finding a stable solution for the power flowing in the network, by adjusting load and generation. Finally the Power Cascade Agent implements some more measurements, for example the number of islands and isolated nodes and the load loss.

### 3.2.2 Measurements

In the last chapter we touched on the topic of measurements already by looking at how the Benchmark Agent is implemented. Here we will explore in more detail how one could implement

new measurements, either replacing or by reusing the existing ones. Two methods are designed especially to handle measurements:

1. public void performInitialStateOperations()
2. public void performFinalStateOperations()

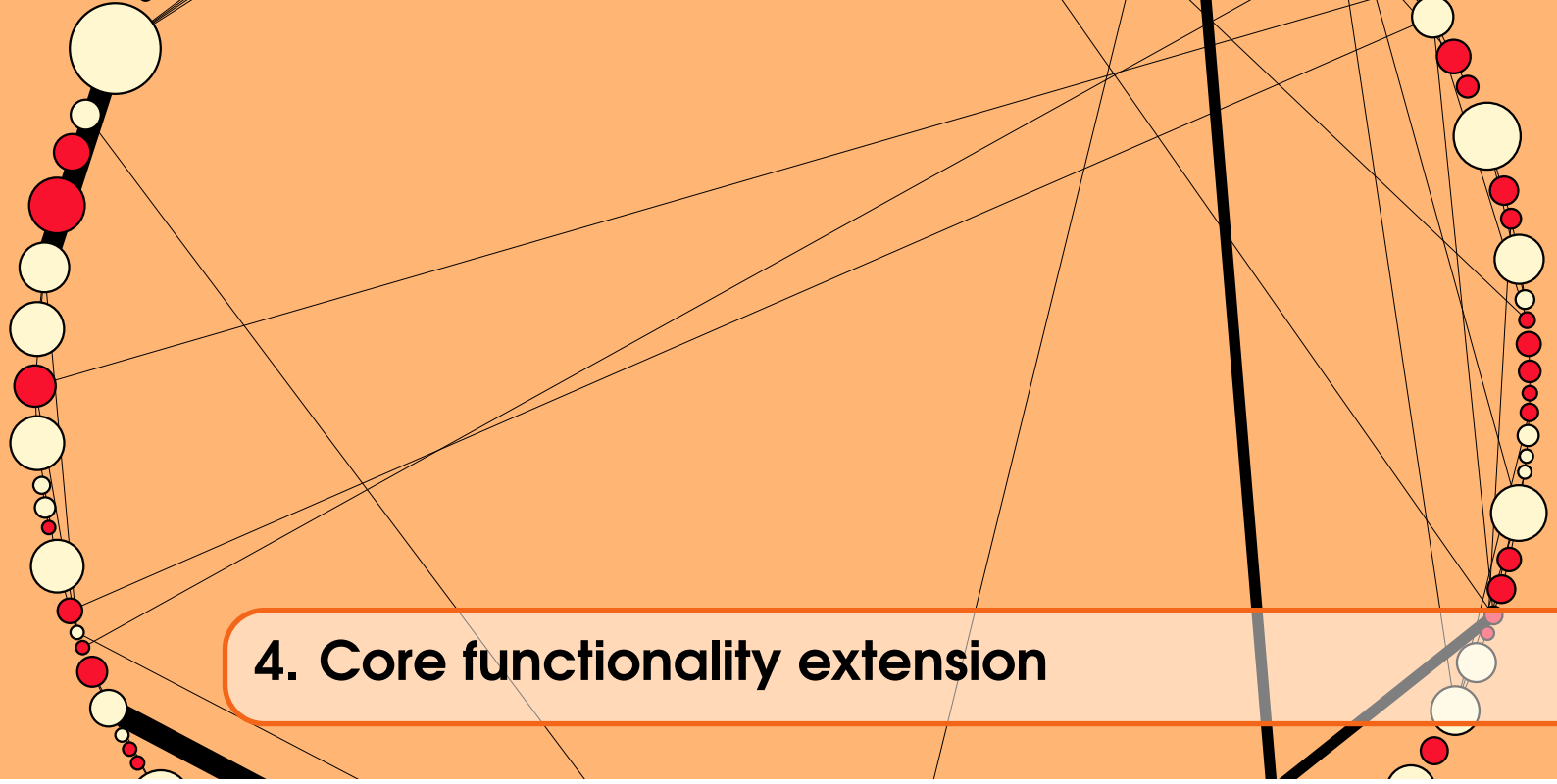
They are executed before and after the main flow analysis respectively, and allow to save values during the current time step, for example into one of the measurement variables temporalLinkMetrics, temporalNodeMetrics or temporalSystemMetrics. To make this more clear, let's take again the Benchmark Agent as an example:

```
public void initMeasurementVariables(){
    HashMap<String,HashMap<Metrics,Object>> linkMetrics=new HashMap<>();
    for(Link link:this.getFlowNetwork().getLinks()){
        HashMap<Metrics,Object> metrics=new HashMap<>();
        linkMetrics.put(link.getIndex(), metrics);
    }
    this.getTemporalLinkMetrics().put(
        this.getSimulationTime(), linkMetrics);
    // initialization of other measurement variables
}

public void calculateFlow(){
    for(Link link:this.getFlowNetwork().getLinks()){
        double flow=link.getFlow();
        HashMap<Metrics,Object> metrics =
        this.getTemporalLinkMetrics().get(
                this.getSimulationTime()).get(link.getIndex());
        metrics.put(
                Metrics.LINE_FLOW, (link.isActivated()) ? flow : 0.0);
    }
}

@Override
public void performInitialStateOperations(){
    this.initMeasurementVariables();
}

@Override
public void performFinalStateOperations(){
    this.calculateFlow();
    // more measurement methods
}
```

Here you see the two overwritten measurement methods introduced above, which now include new methods performing measurement tasks. The first one just initializes the needed variables to hold the measurement data. The second one gets the current flow through every line and saves it for later use. As long as these measurements are implemented in their own public methods, such as the calculateFlow() method in the above example, they can also be used by other Agents, which further extend the current one.

Measurements in SFINA are using the Protopeer framework (see section 2.5 for more details). At the end of each time step, information from the variables introduced above can be saved for later use. This is done in the method scheduleMeasurements() by using log.log(int epoch, Object tag, double value), which assigns to each value the epoch (time step) in which the measurement takes place and one or multiple tags. This information is then saved to a binary serializable file on the hard drive.

The serializable object can be loaded after the simulation finished to perform further calculations and examination The epoch number and tags are used to retrieve the measured values. Additionally Protopeer provides handy methods to do calculations on the data, such as statistics, calculating the mean, retrieving the maximum value, etc. An implementation of this procedure can be seen in the BenchmarkLogReplayer application.



# 4. Core functionality extension

## 4.1 New backend

Backends are software modules that implement algorithms for the calculation of the flow in the network. Often simulation software already exist for various domains, for example MATPOWER or InterPSS to perform power flow analysis. Three ways to implement new backends:

1. Implementing the flow calculation from scratch, using the SFINA data stored in the flow network, nodes and links. This is the most straight-forward way, and definitely the cleanest, resulting in a fully integrated backend. Depending on the complexity however, this can be quite a big task.
2. Integrate an existing backend which was written in Java. This was done for example for InterPSS. First the SFINA data has to be translated to the new backends specific format, then its power flow algorithm is called and finally the data is translated back to SFINA. It can be tedious to make sure the data is translated in the correct way, but potentially providing an easy way to integrate a new backend.
3. Integrate an existing backend written in another language. MATPOWER, which is written in Matlab, was integrated that way. The approach of 2. applies here as well, however on top an interface between Java and the other language has to be developed or integrated. In the Matlab case a package called matlabcontrol written by a third party was used to call Matlab and pass commands to it.

Any backend implements the Backend Interface with the some core methods. Therefore its skeleton has the following structure:

```
public class SomeBackend implements FlowBackendInterface{

    private FlowNetwork net;
    private boolean converged;
    private static final Logger logger =
        Logger.getLogger(SomeBackend.class);

    public SomeBackend(HashMap<Enum, Object> backendParameters){
        this.converged=false;
        setBackendParameters(backendParameters);
    }

    @Override
    public void setBackendParameters(
        HashMap<Enum, Object> backendParameters){
        // parameters for this backend can be
        // retrieved from the HashMap and set here
    }

    @Override
    public boolean flowAnalysis(FlowNetwork net,
                HashMap<Enum, Object> backendParameters){
        setBackendParameters(backendParameters);
        return flowAnalysis(net);
    }

    @Override
    public boolean flowAnalysis(FlowNetwork net){
        // the actual flow analysis is computed here
        return converged;
    }
}
```

Any backend has to implement the FlowBackendInterface, which one can see in the first row. If there are necessary parameters for the computation of this specific backend, they can be passed through the backendParameters HashMap, either to the constructor or directly calling the flowAnalysis() method. The actual computation happens in the latter method. Finally the flowAnalysis method of the backend should return a boolean value specifying if it found a stable solution for the flow in the network based on the given input, i.e. if it converged.
The next step is to adjust the loaders, so it knows how to deal with the new backend. For this it is necessary to modify the following in the input package (replacing Domain* by the current domain the backend is made for):

1. Add the new backend as an enum type variable to Domain*Backend.java. Choose a name for it that is unique and easy to understand
2. Add the new backend to SfinaParameterLoader.java and choose a string for the name, which should be used to initialize experiments, i.e. written to the configuration files in order to use this backend.
3. Add the new backend to the getActualSystemValue(...) method in EventLoader.java. The same string like in SfinaParameterLoader should be used. This makes sure, that the backend can be changed during runtime by events.
4. If backend specific parameters are necessary for the new backend (passed to it by the

backendParameters HashMap, see above for explanations), then they have to be added to Domain*BackendParametersLoader.java. The necessary Enum variables can be specified similarly to PowerBackendParameter.java. Now they can be loaded from the backendParameters.txt file during initialization of experiments.

5. Last of all, the new backend has to be added in SimulationAgent.java to the method callBackend(FlowNetwork net) for the corresponding domain. Now it is called automatically, when specified in the sfinaParameters.txt configuration file.

### 4.1.1 Overview of necessary adjustments

To summarize, the following methods in the SFINA architecture have to be modified in order to add a new backend:

| Package | Class | Method/Explanation |
|---|---|---|
| domain*.backend | Create a new class for the backend | all |
| | Domain*Backend.java | Add enum for new backend |
| | Domain*BackendParameter.java | Optional, if backend parameters are necessary |
| domain*.input | Domain*BackendParametersLoader.java | |
| input | SfinaParameterLoader.java | loadSfinaParameters(args) |
| | EventLoader.java | getActualSystemValue(args) |
| core | SimulationAgent.java | callBackend(args) |

Table 4.1: Necessary adjustments for implementing a new backend. * replace with name of the corresponding domain.

## 4.2 New domain

Currently only some domains are supported, but we envision the extension of SFINA to more domains like transportation, information, gas or general topological diffusion networks. How would one go about integrating a new domain? This is a task which requires changes outlined in the following section.

### 4.2.1 Data

First of all, a new domain requires specific data in order to be able to calculate the flow/distribution of some quantity in the network for given conditions. Therefore one has to decide, which data is necessary in addition to the topological structure of nodes and links. This data then has to be written in the SFINA file format, either by converting it from other data sources or by creating your own test case files (more information in section 2.1 about the file system).

### 4.2.2 Backend

When the data is ready, the next step is to extend the code such that it can load the new flow information. The topological information is general and its loading is implemented already independently of the domain. To load the flow information, a flow loader has to be created for the domain having the following structure (calling the domain now “NewDomain”, which should be replaced by the corresponding domain name):

```
public class NewDomainFlowLoader {
    private final FlowNetwork net;
    private final String parameterValueSeparator;
    private final String missingValue;
    private static final Logger logger =
```

```
        Logger.getLogger(NewDomainLoader.class);

    public NewDomainLoader(FlowNetwork net,
                String parameterValueSeparator,
                String missingValue){
        this.net=net;
        this.parameterValueSeparator=parameterValueSeparator;
        this.missingValue=missingValue;
    }

    public void loadNodeFlowData(String location){
        // implement loading here
    }

    public void loadLinkFlowData(String location){
        // implement loading here
    }
```

The constructor and necessary private variables in the above code example are generic for any flow loader and can be used as shown. What has to be implemented for the new domain are the two methods loadNodeFlowData(...) and loadLinkFlowData(...). Create two new enum list java classes called NewDomainNodeState.java and NewDomainLinkState.java (replacing "NewDomain" accordingly) and add enum types for each flow information, which is necessary for the nodes and links. Then in the two methods in the flow loader, load the flow data and add it to the links and nodes in the following way:

```
if(!value.equals(this.missingValue))
        node.addProperty(NewDomainNodeState.NewNodeState, value)
```

Where NewNodeState is one of the enum type that was created before and value is the value we got from the file corresponding to this node state. We only add this state to the node, if the value is not the missing value string, as is explained in section 2.2. The same procedure applies for the links. It is a good idea to follow the same procedure used in the PowerFlowLoader.java.
Now we have the data loaded and available for running our experiments. To complete the handling of the new data, it is also necessary to create a class that writes the data again in the same format to files. Just follow the PowerFlowWriter.java example to do so.

### 4.2.3 Integrate the domain into the framework

Next, a new domain needs a backend, which uses the data that we now have at hand to compute the distribution or evolution in the network for the giving constraints. This is already explained in detail in section 4.1 on how to integrate a new backend.
Having the data and the backend, most of the work is done. As a last step, it is necessary to integrate the domain in several spots in the framework, in order to make it available for using (again summarized in the table further below):

1. Add the new domain as an enum variable to Domain.java.
2. Modify in SFINA Agent the method
   (a) loadData(): Add the domain to the switch(domain) statement and create an instance of the new flow data loader
   (b) setFlowParameters(): Add the domain to switch(domain) and define which flow variables define flow and capacity
   (c) outputData(): Same as for loadData()
3. Add the new domain to the loadSfinaParameters() method in SfinaParameterLoader.java
4. In EventLoader.java

(a) add the new domain to the “case FLOW” part and create a new lookupDomainNodeState() and lookupDomainLinkState() method with the new flow information
(b) add the new domain to the method getActualSystemValue() and assign a string name to it, which can be used to define events in events.txt

Congrats, now the new domain is ready to be used in applications (section 3.1) and experiments (section 1.2).

### 4.2.4 Overview of necessary adjustments

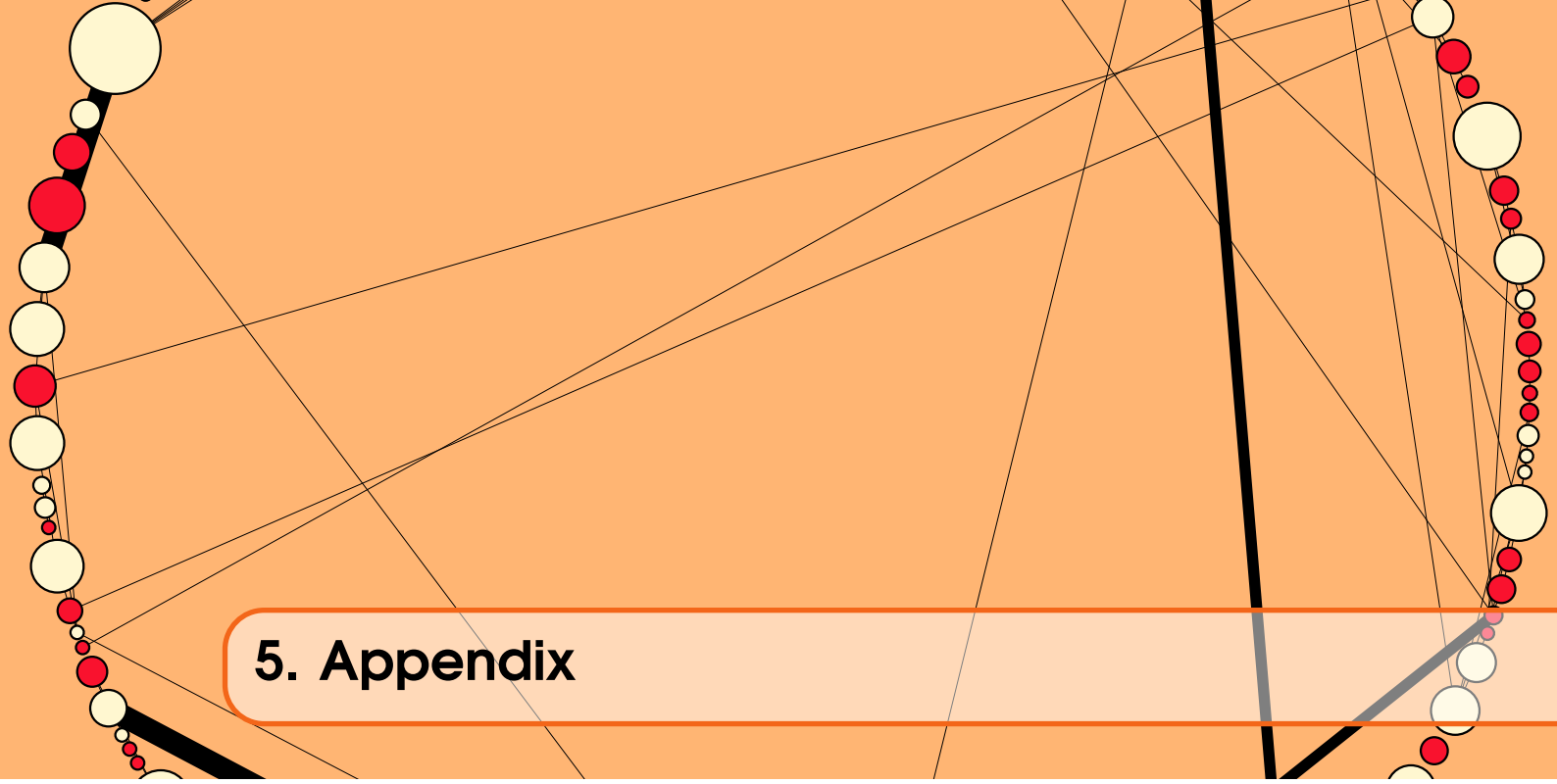
In addition to the adjustments already explained in the section about creating a new backend, the following table gives an overview for implementing a new domain:

| Package | Class | Method/Explanation |
|---|---|---|
| input | Domain.java | add enum |
| | SfinaParametersLoader.java | loadSfinaParameters() |
| | EventLoader.java | - loadEvents()<br>- lookupDomain*NodeState()<br>- lookupDomain*LinkState()<br>- getActualDomain*NodeValue()<br>- getActualDomain*LinkValue() |
| domain*.input | Domain*FlowLoader.java | all |
| | Domain*LinkState.java | |
| | Domain*NodeState.java | |
| domain*.output | Domain*FlowWriter.java | |
| core | SimulationAgent.java | - loadData()<br>- setFlowParameters()<br>- outputData() |

Table 4.2: Necessary adjustments for implementing a new domain. * replace with name of new domain.

# III

# Appendix



# 5. Appendix

## 5.1 Glossary

**flow network**
A mathematical graph with physical properties. A collection of nodes connected by links through which (conserved or nonconserved) quantities flow.

**node**
A point in the network to/from which links point and which can have any number of properties, defining its behaviour.

**link**
A connection between two nodes, having a defined direction from a start node to an end node. Can have any number of properties, defining its behaviour.

**topology**
The network structure created by nodes and links.

**flow**
Data about the quantities flowing through the flow network, i.e. through the nodes and links.

**domain**
The general physical setting of the flow network, defining which quantities of the nodes and links are necessary in order to compute the flow through the network. Examples: Electrical power, transportation, information, ...

**backend**
Code that can compute the flow through the nodes/links if the necessary values for its domain are provided. Can converge or not converge, whose meaning has to be understood physically.

#### event

A change in the flow network, or its nodes/links respectively.

1. Event **feature**: Abstract notion of the network part (topology, flow, system).
2. Event **component**: Node or link.
3. Event **parameter**: Defining what is to be changed.
4. Event **value**: New value for the corresponding parameter.

#### agent

Implementation of SFINA functionality and its applications.

#### time step

Implementation of SFINA functionality and its applications.

#### simulation time

...

#### epoch

...

#### iteration

...

## 5.2 Useful flow network methods

| Names | Description |
|---|---|
| getNode("id") | Retrieve a node from the network by their id |
| getLink("id") | Retrieve a link from the network by their id |
| getNodes() | Get all nodes. Returns a collection |
| getLinks() | Get all links. Returns a collection |
| activateNode("id"), deactivateNode("id") | Activate/deactivate node by their Id |
| activateLink("id"), deactivateLink("id") | Activate/deactivate link by their Id |
| computeIslands() | Extract disconnected components. Returns ArrayList of flow networks |
| getShortestPath(Node a, Node b) | Computes the shortest path between two nodes |
| getDegreeDist() | Returns a LinkedHashMap of node degree vs number of nodes |
| getClustCoeff() | Compute clustering coefficient. Returns double value |
| getAvgNodeDegree() | Compute average node degree. Returns double value |
| getClosenessCentrality(Node node) | Computes node closeness centrality |
| getDegreeCentrality(Node node) | Compute node degree centrality |

## 5.3 Useful nodes methods

## 5.4 Useful links methods

| Names | Description |
|---|---|
| getIndex() | Returns the index of node |
| getLinks() | Returns a collection of links |
| isActivated() | Returns a boolean weather a node is operational or not |
| isConnected() | Returns a boolean weather a node is connected or not |
| getIncomingLinks() | Returns all the incoming links |
| getOutgoingLinks() | Returns all the outgoing links |
| getCapacity() | Returns the capacity of the node |
| setCapacity() | Sets the capacity of the node |
| addLink("id") | Adds a link specified by their id |

| Names | Description |
|---|---|
| getIndex() | Returns the index of link |
| isActivated() | Returns a boolean weather a link is operational or not |
| isConnected() | Returns a boolean weather a link is connected or not |
| getStartNode() | Returns the start node of the link |
| getEndNode() | Returns the end node of the link |
| getCapacity() | Returns the flow capacity of the link |
| setCapacity() | Sets the capacity of the link |
| getFlow() | Returns the double for flow in the link |
| setFlow() | Sets the flow of the link to the assigned value |

# Index

## N



## O



## P



## T