The Informal Guide to the Virtual Network Embedding MIP Creator

TU Berlin, Germany

Matthias Rost
mrost@inet.tu-berlin.de

What is the VNetEMC?

The Virtual Network Embedding MIP Creator (VNetEMC) is a framework for creating Mixed-Integer Programs (MIP) to solve the Virtual Network Embedding Problems (VNEP) while allowing for a large variety of types of models. The VNetEMC supports the modeling of

- (un)directed substrate networks with (un)capacitated nodes and (un)capacitated links,
- three main different flow models: splittable flow, unsplittable flow and confluent flow,
- further extensions to e.g., (dis)allow the collocation of virtual nodes or support for performing access control (i.e., selecting a "good" subset of virtual networks).

As the VNetEMC can be easily extended, a multitude of objective functions can be implemented.

By using XML data representation and exporting MIP models as human-readable GNU MathProg modeling language (GMPL) models, the VNetEMC allows for the concise presentation of computational results and ensures a high confidence in the correctness of the results.

Besides the creation of GMPL models and the derivation of executable `.lp` files, we provide a basic API to generate VNEP scenarios together with scripts to execute them.

The VNetEMC provides for an own XML solution format, such that raw solutions for the `.lp` files (generated e.g.,by Gurobi) can be again parsed to a human-readable format. This not only allows for a verification of the correctness of the results (and the models), but also enables the dynamic creation of new scenarios given solutions to previous ones.

Contents

Part I.

Theory

1. Supported Embedding Models

In this section we will shortly outline the supported embedding models, i.e. which variants of the Virtual Network Embedding Problem (VNEP) the VNetEMC supports.

1.1. Substrate and VNet Model(s)

The literature on the VNEP considers multiple different *types* of substrates and virtual networks. The main dimensions are the following:

Directed vs. Undirected Substrates As the VNEP is a graph-theoretic problem, some researchers consider directed or undirected substrate networks. We support both these models.

Directed vs. Undirected VNets Analogously, one may consider directed and undirected VNets. However, as we argue below, undirected VNets are rather uncommon, and we hence do not support undirected VNets.

Node and Link Resources While nodes and links may have multiple different resource types (e.g., storage, CPU, bandwidth, latency, . . .), we consider only a single node and a single link resource. Additional resources could however be introduced easily.

Capacitated vs. Uncapactiated Substrates Early work on the VNEP typically considered the load minimization problem, without enforcing capacity constraints on substrate nodes and substrate links. We therefore allow for both (un)capacitated substrate nodes as well as (un)capacitated substrate links.

Capacitated vs. Uncapacitated VNets In contrast, nodes and links of VNets must always be attributed with a requested capacity. While it would be generally possible to allow for uncapacitated nodes or links, we argue that this would make little sense, as the core reason to use the concept of VNets is the ability to specify the amount of resources requested.

1.2. Semantics of (Un)directed Substrates and (Un)directed VNets

To better understand the semantics of (un)directed topologies in the VNEP, we will first discuss the semantics of (un)directed substrate topologies:

Directed Substrate Using directed substrate networks, each link can be considered as a single simplex channel between nodes with an own capacity. This directed model is appropriate in most scenarios, as the vast majority of (wired) links are full-duplex and therefore represent two independent simplex lines. Furthermore, using only directed links, we can model that a substrate node s_1 may communicate with s_2 (by including link (s_1, s_2)) while s_2 may not communicate with s_1 (by not including the link (s_2, s_1)). This may be applicable e.g.,in wireless transmissions where s_1 is a powerful sender while s_2 is not as powerful.

Undirected Substrate In contrast, in undirected substrates an edge $\{s_1, s_2\}$ indicates that both connected nodes may communicate with each other. If furthermore a capacity for an undirected link is given, then we must understand the undirected substrate edge as a *shared* resource: while s_1 may send data towards s_2 and s_2 may send data towards s_1, the sum of exchanged data must not exceed the links capacity. Therefore, the undirected substrate network (with capacities) can be understood as a fully directed network, i.e., each undirected edge is represented using two opposingly directed edges, in which the flow along the pair of opposingly directed edges is upper bounded.
Undirected substrates may therefore be used to model e.g.,radio links (using the same frequency)

In contrast to the above discussion, we will now argue that modeling undirected VNets explicitly makes only little sense. We assume that by using an undirected virtual link $\{v_1, v_2\}$ with capacity c, a duplex connection is modeled such that both nodes can communicate with each other (in parallel) such that the total amount of flow sent between the nodes is less than or equal to c.

Undirected VNets on Undirected Substrates In this setting, representing the undirected virtual link $\{v_1, v_2\}$ as directed link (v_1, v_2) suffices: as the substrate is undirected, embedding the directed virtual link (v_1, v_2) will allocate resources on *both* "directions" of used undirected substrate edges. Therefore, any flow in the substrate between v_1 and v_2 of capacity c will readily allow for duplex communication of capacity c, such that v_1 may send flow f_1 towards v_2 while v_2 may send flow f_2 towards $v1$ as long as $f_1 + f_2 \leq c$. Therefore, this case does not need to be modeled explicitly.

Undirected VNets on Directed Substrates Under the assumption that an undirected virtual link should be embedded as a single entity, embedding undirected virtual links on directed substrates may not be possible: assume that the undirected virtual link is to be embedded on a ring substrate (with clockwise links only), then no single path connecting v_1 and v_2 allows for the simultaneously communication of v_1 with v_2 *and vice versa*. Therefore, in this setting, undirected virtual links need to be represented using the two directed links (v_1, v_2) and (v_2, v_1). Assume that (v_1, v_2) is embedded by flow f_1 and (v_2, v_1) is embedded by f_2 and that the capacity of $\{v_1, v_2\}$ is c, then on any directed substrate edge e used by f_1 and f_2, the maximal allocations needed are $\min(f_1(e) + f_2(e), c)$. While such a flow model could indeed be introduced, we conclude that in this setting two distinct virtual links are necessitated such that we do not allow for undirected VNets.

1.3. Model Extensions

While the VNetEMC allows for multiple substrate and flow types, we also include the following further concepts.

1.3.1. Access Control

The task of the VNEP can be coarsely subdivided into access control and load balancing objectives. In access control scenarios (with hard substrate capacities), the main question is “which VNets shall be embedded?”. On the other hand, given a set of VNets that are either known to be embeddable or not considering substrate capacities, the other main question is “how can the provider optimize the allocations?”. With respect to this question, the objective can be e.g.,to minimize substrate allocations, or to spread the load the best possible way, or to save energy by disabling substrate links and nodes.

All our models generally include access control; however, for each VNet a flag can be set, indicating that the VNet must be embedded. Therefore, both access control and allocation optimization scenarios (and mixtures thereof) can be modeled.

1.3.2. Disjoint Node Mappings

Most literature on the VNEP considers only node mappings with disjoint node mappings (per VNet), such that given a VNet all virtual nodes must be mapped to distinct substrate nodes. While this may be sufficient in wide-area scenarios or when explicit replication is requested, mapping multiple virtual nodes on the same substrate node in a data center makes sense, as e.g.,substrate nodes may represent a whole server rack with (unlimited) IPC capabilities. In a recent paper by Carlo FÃijrst and Stefan Schmid, it was also shown that such collocation of virtual nodes may significantly reduce resource allocations in the substrate as connected nodes that are collocated do not require bandwidth allocations on links.

To handle both cases, we again allow to specify a flag, indicating whether disjoint node mappings are required.

1.3.3. Link Groupings Optimization

All our flow models (see next section) make use of so called link groupings to potentially reduce the number of variables and constraints used significantly. To understand the concept, consider a directed star VNet consisting of one center node v_0 and n nodes connected to it $\{v_1, ..., v_n\}$. While “standard” MIP formulations would introduce flow variables for each of the n links, only one set of flow variables suffices: each multi-commodity flow with n senders and a single receiver can be easily modeled by a standard $s - t$ flow by introducing an (imaginary) super source o which is connected to all nodes around the center. By setting the capacacity of (o, v_i) to the capacity of the original link $c(v_i, v_0)$ and requiring o to send $\sum_{i=1}^{n} c(v_i, v_0)$ much flow, this reduces to a simple $s - t$ flow problem.

Similarly, the same holds for an outgoing star where the center is connected to all other nodes: we introduce an (imaginary) super sink and connect all nodes (except the center) to it. Using essentially construction as above, again n flows can be represented using a single flow.

This optimization can therefore reduce the number of flow variables by as much as the number of virtual links. The worst case however is the directed cycle: no matter how links are "grouped", one always needs as much link groups (and therefore flows) as links.

1.3.4. Node Placement Restrictions

Especially when applying the VNEP in the wide-are setting, e.g.,for optimal service-deployment, it is reasonable to assume that virtual nodes may not be mapped arbitrarily, but virtual nodes must be mapped on *regions* of substrate nodes, such that e.g.,all customers within some distance can be served by any substrate node within the region.

Another reason to consider node placement restrictions is the simple fact, that the VNEP is exceptionally hard when the substrate is large and virtual nodes can be mapped on any substrate node.

Again, to be as general as possible, node placement restrictions can be defined for each virtual node independently. If no such restrictions are defined, then the virtual node can be mapped on any substrate node.

1.4. Objective Functions

While we shortly mentioned possible objectives for the VNEP, we have implemented only a single one (, yet). The single objective currently implemented is to minimize migration costs: For each virtual node v and each substrate node s (that v can be mapped on) a migration cost $c(v,s) \geq 0$ is given. The task is to find an embedding minimizing the sum of incurred migration costs. This objective could e.g.,be used to study the benefits of allowing migrations:

- Initially VNets $\mathcal{V} = \{V_1, V_2, ..., V_n\}$ are embedded upon the substrate.
- Given some new set of VNets $\mathcal{V}' = \{V_{n+1}, ..., V_{n+k}\}$, we can now answer the following question: how many migration costs are necessary to embed all of the k new VNets (if all can be embedded)?

As developing, investigating and comparing further objectives is of high interest, we detail in Chapter 7 how further objectives can be introduced to the VNetEMC.

1.5. Flow Models

While all formulations for the VNEP assume the same mapping model of virtual nodes onto substrate nodes (namely: each virtual node needs to be mapped), embedding virtual links can be done in several ways:

Splittable Flow This is the arguably most general model to embed virtual links: each virtual link can be mapped on a set of paths, such that each path carries some (arbitrary amount of) flow; the sum of these flow values must yield the required bandwidth.

Unsplittable Flow In this model, each virtual link must be embedded as a single path, such that on each substrate link along the path the whole bandwidth of the virtual link needs to be reserved.

Unsplittable Uniform Flow Assuming that all virtual links have the same requested bandwidth, the unsplittable flow model can be described using less binary variables. Therefore, we introduce an own flow type for this optimized version of unsplittable flows.

Confluent Flow per VNet While in the unsplittable flow model, each virtual link can be mapped using an arbitrary path, the confluent flow model (per VNet) requires that the flows are *routable* using routing tables (for each VNet):

- Each substrate node holds a simple routing table for each substrate destination and each VNet.
- Flow destined for a substrate node may only be forwarded on a single link (for each VNet).
- The above implies, that for each substrate node destination (for each VNet) there exists a routing tree, such that flow must be routed along it.

Globally Confluent Flow The same as the confluent flow per VNet model, but the routing tables are not multiplexed using the VNets, but are global: For each substrate destination, there exists a single routing in the substrate network, such that all flow (from all VNets) destined to the same substrate node must follow the same routing tree.

The above introduced flow models, correspond to different degrees of virtualization support for links. The globally confluent flow model presents standard routing (e.g., IP forwarding based on destination address prefixes) without any virtualization support. The per-VNet confluent flow model could represent a VNet based on IP protocol stacks. Unsplittable flow models are appropriate in MPLS or OpenFlow contexts. Lastly, the most general model of splittable flow, could (hypothetically) be realized by using IP-based VNets with a Multipath-TCP transport layer.

In the next chapter, we compare the "complexity" of the different flow models by means of comparing the number of variables needed.

2. Complexity of Models

In this chapter we consider the complexity of the VNEP formulations as introduced in the above chapter. As complexity measure, we use the number of linear, binary and integer variables and the number of constraints.

Even though we allow for numerous possible models, all of our models only introduce *as many* constraints and variables *as necessary*. There exists however one case, in which more variables and constraints are generated than necessary: for some flow models we introduce flow allocation variables that are essentially not needed. We however include them such that in each model flow allocations along edges can be accessed (within the models and also from the outside, when reading the solver's solution files) in an uniform fashion. Importantly, as these flow allocation variables are set (with an equality constraint) the solver's presolver will (at least in the test instances) substitute these variables with their defining term and remove the equality constraints.

2.1. Notation

$|E_V|$ denotes the *total* number of *virtual* links, $|E_S|$ denotes the number of substrate links, $|V_V|$ denotes the *total* number of virtual nodes and $|V_S|$ denotes the number of substrate nodes; $|\text{Groups}(E_V)|$ denotes the *total* number of link groups needed to represent the virtual links, and $|\text{Groups}^+(E_V)|$ indicates that only link groups with a common virtual destination are considered; $|\mathcal{V}|$ denotes the number of VNets, $|\text{dest}(V_V)|$ denotes the number of virtual nodes with incoming links and $|\text{dest}(V_S)|$ denotes the overall number of substrate nodes that these virtual nodes can be mapped on.

2.2. Complexity of the Basic Model

We consider in the following the basic model without any extensions. To generate a feasible model for the VNEP a flow model must be included.

Variable Type	**# Binary**
Deciding whether requests are embeddeed	$|\mathcal{V}|$[a]
Guarantee node mapping	$|V_V|$

[a] If access control is disabled, then these are set to 1 and can be removed by the presolver

Constraint Type	**# Constraints**
Node mapping	$|V_V| \cdot |\text{dest}(V_S)|$

2.3. Complexity of Extensions

2.3.1. Substrate Node Capacities

No additional variables are needed.

Constraint Type	**# Constraints**
Bounding allocations on substrate nodes	$\|V_S\|$

2.3.2. Substrate Link Capacities

No additional variables are needed.

Constraint Type	**# Constraints**
Bounding allocations on substrate link	$\|E_S\|$

2.3.3. Access Control

No additional variables are needed..

Constraint Type	**# Constraints**
Disabling Access control	$\|\mathcal{V}\|$[a]

[a]Should be removed by the presolver.

2.3.4. Disjoint Node Mapings

No additional variables are needed.

Constraint Type	**# Constraints**
Requiring Disjoing Node Mappings	$\|\mathcal{V}\| \cdot \|V_S\|$

2.3.5. Node Placement Restrictions

No additional variables or constraints are needed.

2.4. Complexity of Flow Models

Table 2.1 presents the number of variables necessary for representing flows while Table 2.2 displays the number of needed constraints. Note that for the unsplittable, the globally confluent and the confluent per VNet flow models, the presolver will (probably) remove introduced variables and constraints.
In Table 2.1 we have ordered the flow models according to their expected hardness: the splittable flow model can be solved in polynomial time, while confluent flow models require not only a large number of binary but also a large number of linear variables.

Considering the number of needed constraints, the complexities are similarly ordered. However, the confluent models introduce an enormous amount of additional constraints.

Flow Model	**# Linear**	**# Binary**	**# Integer**
Splittable Flow	$\lvert\text{Groups}(E_V)\rvert \cdot \lvert E_S\rvert$	0	0
Unsplittable Uniform Flow	0	0	$\lvert\text{Groups}(E_V)\rvert \cdot \lvert E_S\rvert$
Unsplittable Flow	$(\lvert\text{Groups}(E_V)\rvert \cdot \lvert E_S\rvert)$[a]	$\lvert E_V\rvert \cdot \lvert E_S\rvert$	0
Globally Confluent Flow	$\big(\lvert\text{Groups}^+(E_V)\rvert \cdot \lvert E_S\rvert\big)$[a] $+$ $\lvert\text{dest}(V_V)\rvert \cdot \lvert\text{dest}(V_S)\rvert \cdot \lvert E_S\rvert$	$\lvert\text{dest}(V_S)\rvert \cdot \lvert E_S\rvert$	0
Confluent Flow per VNet	$\lvert\, \big(\text{Groups}^+(E_V)\rvert \cdot \lvert E_S\rvert\big)$[a] $+$ $\lvert\text{dest}(V_V)\rvert \cdot \lvert\text{dest}(V_S)\rvert \cdot \lvert E_S\rvert$	$\lvert\mathcal{V}\rvert \cdot \lvert\text{dest}(V_S)\rvert \cdot \lvert E_S\rvert$	0

[a]These variables should be removed in the presolver of the solver.

Table 2.1.: Number of needed {linear, binary, integer } variables for representing flow models.

Flow Model	**# Constraints**
Splittable Flow	$\lvert\text{Groups}(E_V)\rvert \cdot \lvert E_S\rvert$
Unsplittable Uniform Flow	$\lvert\text{Groups}(E_V)\rvert \cdot \lvert E_S\rvert$
Unsplittable Flow	$(\lvert\text{Groups}(E_V)\rvert \cdot \lvert E_S\rvert)$[a] $+ \lvert E_V\rvert \cdot \lvert E_S\rvert$
Globally Confluent Flow	$\big(\lvert\text{Groups}^+(E_V)\rvert \cdot \lvert E_S\rvert\big)$[a] $+ \lvert\text{dest}(V_S)\rvert \cdot \lvert V_S\rvert +$ $3 \cdot \lvert\text{dest}(V_V)\rvert \cdot \lvert\text{dest}(V_S)\rvert \cdot \lvert E_S\rvert + \lvert\text{Groups}(E_V)\rvert \cdot \lvert\text{dest}(V_V)\rvert \cdot \lvert E_S\rvert$
Confluent Flow per VNet	$\big(\lvert\text{Groups}^+(E_V)\rvert \cdot \lvert E_S\rvert\big)$[a] $+ \lvert\text{dest}(V_S)\rvert \cdot \lvert V_S\rvert +$ $3 \cdot \lvert\text{dest}(V_V)\rvert \cdot \lvert\text{dest}(V_S)\rvert \cdot \lvert E_S\rvert + \lvert\text{Groups}(E_V)\rvert \cdot \lvert\text{dest}(V_V)\rvert \cdot \lvert E_S\rvert$

[a]These constraints should be removed by the presolver of the solver.

Table 2.2.: Number of needed constraints for representing the flow models.

3. Modeling Opportunities

In Chapter 1 we have presented the models that the VNetEMC may create. We will now shortly discuss, how more complex scenarios could be either modeled using the existing model capabilities or how the VNetEMC could be extended.

3.1. Modeling Virtual (Oversubscribed) Clusters

Hose models and virtual (oversubscribed) clusters[1] as a general type of VNets have recently been proposed also in datacenter contexts. In general, virtual (oversubscribed) clusters can be considered as an abstraction from the pure graph based model, as they aim at allowing to model traffic matrices instead of pure fixed capacity topologies.

3.1.1. Virutal Clusters

A virtual cluster $VC(N, B)$ consists of a single virtual switch and a N virtual machines that are connected to the virtual switch with bandwidth B. Therefore, the virtual nodes (machines) can only communicate via the virtual switch (a simple star topology). As full-duplex communications are assumed, the virtual link is connected to each virtual machine with the same capacity of B. In the above model (assuming that there exists an even number of virtual machines), it is therefore possible that the first halve of virtual machines send data to the second one at full link capacity of B, while the second halve sends data to the first one at full link capacity of B. However, if e.g. all but one virtual machines try to communicate with a single virtual machine, then the available capacity at which *all* VMs may send/receive data is limited by B.

3.1.2. Modeling Virutal Clusters

As the virtual cluster topology is a bi-directed star with fixed (maximal) capacities, virtual clusters can be readily be specified by the means provided by the VNetEMC. However, as it may e.g. be required that virtual switches can only be mapped on switches of the substrate, it should be possible to mark virtual nodes and substrate nodes as representing (virtual) switches. As this means only introducing a flag into the data model, this can be done without affecting any other part of the VNetEMC. Given this data model extension, it would then be easy to write a special node placement restriction, such that virtual switches can (a priori) only be mapped on substrate switches.

[1]see e.g. Ballani, H., Costa, P., Karagiannis, T., & Rowstron, A. I. (2011, August). Towards predictable datacenter networks. In SIGCOMM (Vol. 11, pp. 242-253).

However, if we wanted to allow for the mapping of virtual switches onto non-switch substrate nodes as well, then further extensions might be necessary. Assuming e.g. that a substrate node *emulating* the functionality of a (virtual) switch requires a certain amount of node resources (e.g. CPU), the model could be adapted in the following way:

- First the information on virtual / substrate nodes should be accessible in the GMPL model.
- Instead of using the standard node capacity model[2] a custom capacity model could be derived for this scenario, such that substrate switches are not capacitated (requiring of course that general virtual nodes cannot be mapped on substrate switches) but only normal substrate nodes are capacited.

3.1.3. Virtual Oversubscribed Clusters

The concept of virtual oversubscribed clusters extends the concept of virtual clusters by allowing multiple virtual cluster groups that are connected by a tree (of height two) of virtual switches, where the virtual machines may communicate with the virtual switch they are connected to at full speed B, but the bandwidth of these virtual switches to the *root virtual switch* is limited to a fraction of the sum of maximal bandwidths with which the virtual switches connected to it may exchange data. Concretely, in addition to N and B, the Virtual Oversubscribed Cluster specifies an oversubscription factor O.
This concept can be introduced in the same way as the virtual clusters.

3.2. An “Ultimatively” General Substrate Model

While the VNetEMC currently allows for using either only directed or undirected edges and similarly either all substrate nodes / links are capacitated or uncapacitated, the user might wish to consider scenarios where some of the edges are directed and the remaining edges are capacitated or in which only some nodes or links are capacitated. We will now shortly outline how such a general model should be included in the VNetEMC.
First note that undirected edges correspond to two opposingly directed edges. Therefore, undirected edges can already be modeled using directed edges only. Considering the enforcement of capacity constraints on undirected edges using a fundamentally directed substrate, we propose the following approach:

- Given a directed substrate (possibly with capacities on the directed substrate links), the user can specify *additional* capacity constraints for *sets* of substrate links.
- To model an undirected substrate edge, such an additional capacity constraint needs to be added to the scenario.
- Using this approach the existing XML and GMPL data models would not need to be changed, which would require a major revision of the VNetEMC.

[2]See `src/model/gmpl/extensions/SubstrateTypes/Nodes/capacitatedNodes.mod`

Using the above approach, not only a major redesign would be avoided, but this would also allow for modeling shared link capacities in a more *general* fashion. As undirected edges basically represent two directed links with *shared* resources, using the above approach one could e.g. model broadcasting domains in wireless scenarios:

- Assume a substrate of n radio stations that are connected to a single base station.
- As both the base station and the radio stations outgoing capacities are limited e.g. by the technology or the hardware employed, both the links from the radio stations to the base station as well as the links from the base station to the radio stations are inherently bandwidth limited. Therefore, these should be modeled using directed, capacitated links.
- However, assuming that all nodes use the same frequency for communication, there exists a natural capacity on the overall available capacity. Therefore, the sum of flow along both edges directed towards the base station and away from the station, should be upper bounded by a certain (frequency and technology dependent) capacity.

The above concept is not only applicable in wireless scenarios but could also be used to model internal bandwidth limitations of non-blocking switches, as their maximal (internal switching) throughput generally lies below the sum of outgoing links' capacities.
It must be noted that the same general concept of additional capacity constraints can be employed for substrate nodes: by limiting the overall sum of allocations on substrate nodes e.g. power constraints could be modeled.

Part II.

Practice

4. Getting Started

In this chapter we will outline the prerequisites to running VNetEMC and how to run the test instances.

4.1. Contents of the Project

The project-archive should contain the following directories and files:

`build/` Contains Ant-scripts for generating the executable jar files for generating the test instances as well as the `create_schemes.sh` shell script to generate .java files corresponding to our XML data model.

`src/` Contains the complete java source of the VNetEMC.

`doc/` Contains the javadoc documentation of the VNetEMC.

`.jars` Precompiled java executables for generating the test instances.

`.xsd` Definition of our XML data model.

`settings.xml` File in which global and user-dependent configurations are stored.

4.2. Required Software

To successfully run VNetEMC, you will need the following:

GLPK The VNetEMC uses the GNU Mathprog modeling language (GMPL) to define embedding models. Using GLPSOL (a LP / MIP solver) of the GNU Linear Programming Kit these models can be easily converted into `.lp` files that can in turn be read by all MIP solvers, such as Gurobi, CPLEX or SCIP. The GLPK can be obtained from `http://www.gnu.org/software/glpk/` [1]. The GLPK package also contains the gmpl.pdf (in doc/) detailing the GMPL language to specify models.

JDK7 Our project makes slightly use of Java7 and therefore a corresponding JDK needs to be installed.

[1]For the Windows operating systems see `http://winglpk.sourceforge.net/`

(JAXB) We make extensive use of the Java Architecture for XML Binding (JAXB). Normally, JAXB should be contained within the JDK such that no setup is necessary. If this is not the case, the reference implementation can be obtained from `https://jaxb.java.net/` such that the corresponding `.jar` files only need to be included in the build path.

Gurobi / CPLEX While GLPSOL is a fully fledged LP / MIP solver, commercial solvers as Gurobi or CPLEX outperform GLPSOL by magnitudes. To even run the moderately small test instances it is therefore advisable to install either of these commercial solvers. As Gurobi has a very user friendly academic licensing approach[2] and the (current) version 5.6 clearly outperforms (the outdated) CPLEX 11.2 [most current version: 12.6] we are currently recommending to use Gurobi (5.6). However, an evaluation of how well CPLEX 12.6 performs versus Gurobi 5.6 would be interesting.

Operating System Our experience of running commercial MIP solvers under Windows is very limited. Due to the good scripting support under Unix-based operating systems, our framework is rather directed towards these operating systems. However, it might be possible to achieve the same level of scripting automatisms under Windows and the VNetEMC could potentially be extended to better support Windows.

4.3. Setting up VNetEMC

To set up VNetEMC you basically only need to unpack the project archive and edit the settings.xml:

- Set `<PathToStored../>` to some (existing or newly created) directories on your machine. These directories will be per default (and especially for our test instances) be used to store generated models. These directories should be only used by VNetEMC, as potentially hundreds of files will be generated (in sub-directories).

- Set `<GMPLCommand/>` such that the command can be executed. If GLPK is properly installed (tested under Ubuntu 12.04) then the default command should work. If no "make install" is wished, then one can of course specify the path of glpsol by hand. Under Windows the same applies, and the default command should work when using glpsol.exe.

- Depending on the OS and the installation, the parameters `<ExecuteGurobiCommand/>` and `<ExecuteCPLEXCommand/>` may need to be edited. Removing the "| tee -i .." fragment will probably allow to run (at least) gurobi under Windows, if "gurobi.sh" is replaced by "gurobi.bat" (again, assuming that these can be found by the shell the commands are executed in).

- The `<InputScriptForGurobi/>` and `<InputScriptForCPLEX/>` parameters allow to specify input scripts for these solvers, such that parameters can be passed to

[2] Allowing to always use the newest version as well as to obtain unlimitedly many licenses.

these solvers. The given InputScripts are rather of exaplanatory nature and should definitely be adapted when running experiments. However, for the test instances that use Gurobi, the settings does not matter as all instances can be solved within a fraction of a second.

4.4. Running the Test Models

Assuming that the path variables of the settings.xml are set correctly (to existing directories), and that GLPK is set up correctly (with the correct ⟨GMPLCommand⟩ field of the settings.xml), the following "script" can be used to generate the testScenarios, run them and generate solution files:

```
java -jar generateTestRequests.jar
java -jar generateTestSubstrates.jar
java -jar generateTestScenarios.jar

(change directory to where the scenarios where created)

cd testScenarios
bash executeTest.sh

(change directory to VNetEMC root directory)

java -jar generateScenarioSolutionForTestInstances.jar
```

Executing generateTestRequests.jar will create two sets of requests in the directory specified for storing the requests and executing generateTestSubstrates will generate 8 different substrate graphs. Executing generateTestScenarios.jar then binds together the requests and substrates under 5 different flow types, generating 40 scenarios overall (in the directory for storing scenarios). For each scenario (-scenario.xml) a model file (.mod), a data file (.dat) and a (.lp) file should be present. Furthermore, an input script should have been generated for each scenario [3].
Within the testScenarios/ directory in which the scenarios are created, a executeTest.sh script should have been created. Executing this script will solve all scenarios using Gurobi, checking that for each model an optimal solution was found.
Lastly, using the output of Gurobi (as specified in the input script file), by executing generateScenarioSolutionForTestInstances.jar, for each scenario a solution file (-scenario-solution.xml) is generated; allowing to inspect the solution in an human-readable format.

[3]See Chapter A for a complete list of files that are (in general) generated.

5. Design Philosophy

When developing a framework to generate MIPs, one may choose between three different approaches:

- Generate the `.lp` files directly from scratch, i.e. that the whole model is encapsualted within the application.
- Use the API from a solver (e.g.,Gurobi or CPLEX) or the API of a mathematical modeling lanugage (as e.g.,AMPL or GLPK) to generate the model using the abstraction layer of these languages.
- Instead of using an API, represent and export the model as simple text files written in an mathematical modeling language (as e.g.,GMPL or ZIMPL).

Our framework entirely relies on the *textual representation* of models. With respect to the other approaches, this has the following advantages.

Maintainability / Extensibility The VNetEMC relies on a strict separation of models and how corresponding scenarios are generated. As parameters, variables and constraints are defined *externally* and *encapsulated* as model file, the models are humanly comprehensible and can be rather easily checked for correctness. Adding a new feature to the set of model extensions can then be done separately from other parts of the model by simply specifying a model *fragment*.

Consistent Presentation of Results Considering the two other approaches for generating MIPs, our approach taken allows for a consistent presentation of computational results. While the first two approaches only export `.lp` files, that are absolutely non-comprehensible for humans, we export the model and data files together with the `.lp` file. Thus, even in retrospect, the underlying model of the .lp file can easily be grasped. This is a big advantage even over API approaches, as to understand the underlying model the user would have to inspect the source code of the generating application; allowing for as many different models as the VNetEMC is targeting, we believe that even in this setting, the user can not grasp the concise model.

Fast Prototyping As we generate GMPL model files with corresponding data files, we allow for fast prototyping. To test a new functionality, it suffices to take an existing model file and the corresponding data file, extend the model by writing GMPL statements and generate the model. In contrast to extending a model in the API approach, this is far less error-prone.

6. Main Components

Before discussing the main components of the VNetEMC, we strongly encourage the reader to have a look at the source of the executables for running the test instances[1], as they already give quite a good overview over the core functionality provided. Furthermore, we stress that our framework comes with an extensive javadoc documentation, providing a high-level description for each package and each class.

6.1. XML Data Representation

The VNetEMC relies on a pure XML representation of scenarios. While this type of representation initially slows down development of a model, it is easily extensible and even more importantly, *human readable*; in contrast to e.g.,GMPL data files, the scenario files produced by the VNetEMC are easily comprehensible, as they allow for structured representation of data.
Considering the effort necessary to write XML files, we employ JAXB, that automatically generates Java classes representing XML elements according to XML schema definitions. This allows for a concise presentation of the model in the VNetEMC.
The `.xsd` files defining our data model are contained in the root directory of the VNetEMC project.

6.2. Generation of XML Scenarios

The directory `src/generator` contains the *generators* for constructing VNets, substrates and binding them together in scenarios. We make heavily use of interfaces, such that the generation of scenarios using e.g.,a new objective only requires changing a single line of code, namely selecting another `IObjectiveGenerator` implemenation, thereby effectively abstracting from the underlying XML data framework. This package therefore provides an *API* to generate scenarios.
Importantly, by using XML wrappers, VNets and substrates can be created and stored independently of scenarios. Thereby allowing to read them later on to create the same scenario under different flow types, using exactly the same substrates and VNets.

[1]These are contained in `src/test/`

6.3. Converting XML Scenarios to MIP Models

The directory `src/converter/scenario/xmlToGmpl` contains the majority of code related to generating the GMPL model and data files. Considering the generation of GMPL model files, the creation process is considerably easy. The directory `src/model/gmpl` contains a basic model, that is common to all our models, and `src/model/gmpl/extensions` contains a model fragment for each possible extension, e.g.,for splittable flows or capacitated substrate nodes. The classes responsible for generating the model therefore load all necessary extensions and plug them into the common model file via textual replacement of placeholds (in the common model file).
The process of writing data files, while being lengthy, simply parses the contents of the scenario XML file into the GMPL data file format. By using the interface `IModelExtension` for model extensions, these are also allowed to write GMPL data elements independently from each other. This for example allows for the encapsulation of writing parameters necessary to define objective functions.

6.4. Generation of XML Solution Files

Given an XML scenario file, we facilitate the generation of XML scenario solution files according to `scenario-solution.xsd`. These scenario solution files represent the solver's solutions, which is a large mapping of variables to values, in an human readable format and in a *structured* fashion. The solution file especially contains the mappings of nodes and links for each VNet, the resulting overall substrate allocations and routing tables[2].
The usage of XML solution files serves not only the easier representation of results, but also e.g.,computes the substrate allocations that are not explicitly manifested in the solver's solution file. Furthermore, we decompose the flow of link groupings into flows for each contained virtual link.
The code for generating the XML solution files is contained in `src/converter/solution`.

6.5. Other Important Features

6.5.1. FilenameHandlers

To be able to understand and reproduce computational results, we advocate the usage of our FilenameHandlers, as defined in `src/util/fileHandler`. Given a unique scenario name, these FilenameHandlers define unique extensios (or names of) files generated when performing a computational evaluation[3].
Using these name schemes, it is easy to generate computational evaluations that are easily comprehensible, as there exists ()what we call) the "chain of evidence". Given a uniquely

[2]Depending on the flow model, routing tables are either specified for each virtual link, or for each substrate node or for each combination of substrate destinations and VNets.
[3]See Chapter A for an overview

named scenario file, all corredponding (model, data, lp, solution, log, ..) files are generated based on previously generated files without changing them. If, in retrospect, the results of a computational evaluation need to be investigated, all intermediary steps to solve the problem are well-documented.

6.5.2. Startup / Execution Scripting Support

Using the scripting related parameters in `settings.xml`, we allow for exporting script files for starting experiments using Gurobi or CPLEX. Furthermore, the `<InputScriptForGurobi/>` and `<InputScriptForCPLEX/>` templates can be used to pass parameters to these solvers, see the default `settings.xml` file.

7. Extending the VNetEMC: How to introduce a New Objective Function

In this chapter we will shortly outline how to extend the VNetEMC by introducing a new objective. As it is sufficiently simple, we will, as an example, consider the objective of maximizing the profit of the provider by allowing access control:

- We assume that all VNets are not required to be embedded, such that always a feasible solution exists (possibly embedding none of the VNets).
- Each VNet $V \in \mathcal{V}$ is attributed with some profit, that is gained, when V is embedded: $\text{profit} : \mathcal{V} - > \mathbb{R}^+$.
- The objective is therefore $\max \sum_{V \in \mathcal{V}} \text{embedded}(V) \cdot \text{profit}(V)$.

7.1. Writing a GMPL-Model Template

To start off, you should prototype the objective within GMPL. To introduce the extension into the VNetEMC, you should use the extension template `blank_extension_template.mod`[1]. Chapter B contains an overview over the common set of input and outputs avaiable to all GMPL models. Note that our GMPL models as well as the XML schemes use the more general term request instead of VNets. A GMPL model fragment could look like this[2] :

```
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#BEGIN_OBJECTIVE_FUNCTION_PARAMETERS

check: card(RequestsThatMustBeEmbedded) == 0;

param profit{Requests} > 0;

#END_OBJECTIVE_FUNCTION_PARAMETERS
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<


#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#BEGIN_DEFINITION_OBJECTIVE_FUNCTION
```

[1] Contained in `src/model/gmpl/extensions/`

[2] Chapter B contains the definition of globally available sets, parameters and variables that can be used in any model extension.

```
maximize profit:
sum{req in Requests} request_embedded[req] * profit[request];

#END_DEFINITION_OBJECTIVE_FUNCTION
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

Above, the check guarantees that all requests are free to be not embedded[3]. The parameter profit, which mus be specified for all requests, gives the profit and the objective function is straight forward. Store this model fragment as `maximizeProfit.mod`[4].

7.2. Extending the scenario.xsd

Having prototyped the objective, you can introduce it into the XML data model. To do that, introduce the new objective type into `scenario.xsd` as follows:

```
<xs:complexType name="MaximizeProfitType">
 <xs:sequence>
  <xs:element name="ProfitOfRequest" type="xs:decimal" minOccurs="1" maxOccurs="unbounded">
   <xs:attribute name="RequestId" type="xs:string" use="required"/>
  </xs:element>
 </xs:sequence>
</xs:complexType>
```

Having done that, you need to extend the `ObjectiveFunctionChoiceType` definition within `scenario.xsd`:

```
<xs:complexType name="ObjectiveFunctionChoiceType">
  <xs:choice>
    <xs:element name="MinimizeMigrationCosts" type="vnetEMCScenario:[..]">
    <xs:element name="MaximizeProfit" type="vnetEMCScenario:MaximizeProfitType"/>
  </xs:choice>
</xs:complexType>
```

Lastly, to update the automatically generated Java classes within the framework, execute the contents of the script `build/create_schemes.sh`. This should generate the Java class `MaximizeProfitType.java`[5], such that the class `ObjectiveFunctionChoiceType` can now contain our new objective.

7.3. Developing a Generator for the Objective

To enable generating scenarios with the new objective, you have to write a class implementing the interface `IObjectiveGenerator` [6]. The objective function is the last element to be generated when XML scenarios are created. Thus, at the time the generator is called via the function `generateObjectiveFunction(ScenarioType scenario)`, all requests

[3]Note that in our GMPL formulation as well as in our XML data representation, we use the more general term requests instead of VNets, as VNets (in our opinion) only describe the topology and resource requirements, but not e.g.,that nodes must be mapped disjointly.

[4]under `src/model/gmpl/extensions/ObjectiveTypes/`

[5]under `src/model/generated/vNetEMC/scenario/`

[6]This file should be stored under `src/generator/objective/maximizeProfit`.

are already added to the scenario. Therefore, you only need to iterate over the requests contained in the scenario, decide on some profit specification (e.g.,uniform or depending on the number of nodes, or ...) and generate the XML data elements (MaximizeProfitType) accordingly.

7.4. Enabling GMPL Export

Once the new objective is included in the generation process, the GMPL export has to be adapted to allow for exporting the new model fragment `maximizeProfit.mod` as well as to generate the corresponding parameters for the data file.
To enable this, first a class extending the `IModelExtension` interface must be written. As we have specified the model adaption in `maximizeProfit.mod`, you should derive a class extending `ExtensionFromFileContentsWithoutData.java` which will handle exporting the model part[7]. In the constructor of this class, the scenario has to be passed, such that the data of the objective is accessible. By overriding the function `void writeData( ScenarioGMPLDataAppender dataAppender)`, you can then write the GMPL parameters[8]
The main entry point for the creation process of the GMPL models and their data files is the class `GMPLCreator.java` [9]. There, you now only have to extend the existing function `handleObjectiveType()`, which inspects the `ObjectiveFunctionChoiceType` XML element and decides which class implementing the interface `IModelExtension` should be loaded. Therefore, if the new XML entry `MaximizeProfitType` is set, the newly developed `IModelExtension` only needs to be loaded.

[7]This class should be stored under `src/converter/scenario/xmlToGmpl/objectiveFunctions`.
[8] See `MinimizeMigrationCostExtension.java` for an example.
[9]contained in `src/converter/scenario/xmlToGmpl`

Part III.

Appendix

A. Overview on Files Generated by the VNetEMC

`-scenario.xml` The XML file representing a single VNEP scenario.

`.mod` The GMPL model file corresponding to the `-scenario.xml` file.

`.dat` The GMPL data file corresponding to the `-scenario.xml` file.

`.lp` The mixed-integer program file corresponding to the GMPL `.mod` and `.dat` files.

`start_` Input script for the MIP solver to run the `.lp` file.

`.sol` The solution file of either CPLEX or Gurobi corresponding to the `.lp` file.

`.sol.meta` A meta-solution file, detailing runtime, objective value etc. corresponding to the `.sol` file if Gurobi was used.

`.log` The log file generated during the solution process of the `.lp` file.

`-scenario-solution.xml` The XML file representing the solution to the original `-scenario.xml` file according to the solution files `.sol,` `.sol.meta` and `.log`.

B. Common Input and Output of the GMPL Models

B.1. Input

```
###########################################
# DEFINITION OF SUBSTRATE GRAPH
###########################################

set VS;
set ES within VS cross VS;


###########################################
# DEFINITION OF REQUESTS
###########################################
set Requests;
set RequestsWithDisjointNodeMappings within Requests default {};
set RequestsThatMustBeEmbedded within Requests default Requests;

set VReq{Requests};
set EReq{req in Requests} within VReq[req] cross VReq[req];

set AllowedSubstrateNodes{req in Requests, VReq[req]} within VS default VS;

set EReqGroupIds {req in Requests};
set EReqGroup {req in Requests, EReqGroupIds[req]} within EReq[req];

param  virtual_spec_node {req in Requests, VReq[req]} > 0;
param  virtual_spec_link {req in Requests, EReq[req]} > 0;
```

B.2. Output

```
###########################################
# VARIABLES
###########################################

var request_embedded {req in Requests} binary;
var node_mapping {req in Requests, VReq[req], VS} binary;
var flow_alloc{req in Requests, EReqGroupIds[req], ES_all} >= 0;
```