# Javanco, a software framework for optical network modelling and optimization

**Sébastien Rumley, Robert Hendry and Keren Bergman, *Fellow, IEEE***
*Department of Electrical Engineering, Columbia University, 500 W. 120th St., New York, NY 10027, USA*
*e-mail: sr3061@columbia.edu*

**ABSTRACT**
Most studies aimed at designing and optimizing optical networks, involve substantial software development. The software tools resulting from each project are varied but generally have parts in common. Regrouping these overlapping parts in a single framework can be valuable and enable the efficient usage of developed features for new networks design and performance evaluation. In this paper, we present such a framework, Javanco (JAVa Aided Network COckpit), which integrates multiple tools and focuses on graph level functionalities. We compare the Javanco architecture and its benefits with other frameworks such as OMNeT++.
**Keywords**: optical networks, software framework, simulation.

## 1. INTRODUCTION

Research in the field of optical network planning, dimensioning, and performance analysis generally require substantial effort in software development. Network planning and dimensioning (deciding where to put fibres, lightpaths, regenerators, etc.) is achieved by means of algorithms. The performance of these algorithms can sometimes be estimated by analytical methods, i.e. keeping them expressed in pseudo-code. In most cases, however, a software implementation and an application of tangible test cases are required to check their validity and measure their performance. Simulation is also often required to assess the behaviour of dimensioned networks in a dynamic environment, i.e. when random traffic is injected, when network elements fail, when the network configuration is changed, or to validate network adaptation mechanisms [1]. Furthermore, aside from the coding efforts strictly related to the algorithm or the simulator development, data pre- and post-processing procedures are also required. Pre-processing involves preparation of the input data structures taken by an algorithm (in particular the baseline topology), post-processing the analysis of the rough data returned by the algorithm or the simulator.

To reduce the time required to develop this software, a framework approach is generally chosen. Procedures frequently employed are coded separately, once, and are accessed when required by the algorithms and simulators. A framework approach also permits the reuse of code written and validated by third parties. A certain level of homogeneity is however required among the components developed within the framework, and this requirement can itself become problematic. Hence, a framework involving overly strict rules may limit the range of studies that can be implemented within it. The choice of the framework is therefore an important decision to be taken before starting algorithm or simulation implementations [2].

Several frameworks (OPNET, OMNeT++, ns-2 and ns-3) are broadly reported in the literature [1-5]. Most are organised around an event driven simulation kernel enriched by extensions permitting simulation of various devices, protocols, or even a protocol stack (e.g. FTP over TCP over Ethernet). These frameworks are well suited for studies involving such protocols. In the context of optical networking, however, high level protocols such as TCP are rarely used since traffic is mostly considered as a commodity to shift from one place to another. Optical networks also often use ad-hoc protocols that are not commonly available. In this context, the availability of protocol extensions compatible with a given event-driven simulator plays a less important role for optical networks in the framework choice.

In this paper, we present a framework built around functionalities related to graph theory. Indeed, many planning algorithms or simulators use features from this field, in particular for routing. We relate the experience gathered in developing this framework, called Javanco. After presenting its main features in Section II, we detail the various contexts in which it has been used (Section III). We also describe how it differs from other frameworks and discuss the pros and cons associated with this distinction (Section IV). We draw conclusions on the proposed approach in Section V.

## 2. JAVANCO GENERAL DESCRIPTION

Javanco is programmed in the Java 6 language. Java's portability and platform independence played a major role in the programming language choice. Java is also a good compromise between performance and accessibility to new users (comprehensive Application Programming Interface (API), eased memory management). Java finally offers interesting code introspection and profiling capabilities, the latter helping the programmer to better understand the behaviour of his or her programs.

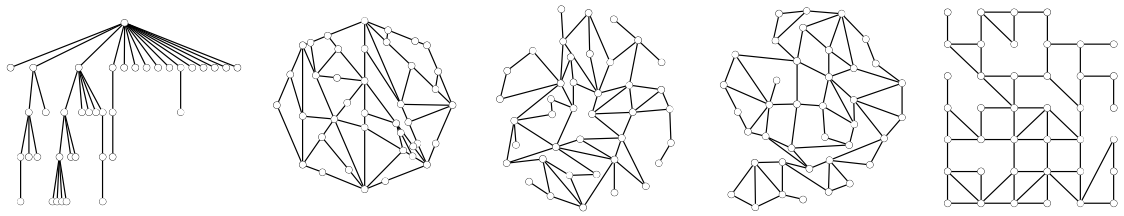

*Figure 1. Examples of random planar graphs generated in Javanco*

### 2.1 Main structure: layers, nodes and links

Javanco is organised around a main java object, the *GraphHandler*, through which nodes and links can be added, deleted, and/or retrieved to or from a graph. A concept of *layers* permits Javanco to have several superposed graphs inside the same *GraphHandler*. This feature can be exploited to segregate the links representing the fibres from the ones representing the lighpaths: they would be regrouped on two different layers, while sharing common nodes as extremities. Nodes and link objects can be tagged with attributes. An attribute is *key=value* pair of character strings. These attributes can be used to define the high-level characteristics of an object, as its capacity, throughput, length, etc. Javanco nodes and links can also host *user objects* for storing complex structure hard to encode with attributes. A typical user object can be a *lightpath* object storing information such as the number of carried tributaries, the occupation, etc. User objects can additionally provide experiment related methods.

### 2.2 Graphical interface

The Graphical User Interface (GUI) libraries form two sub-groups: the ones responsible for the painting of the graph (computer to human interaction – the *view*) and the ones in charge of the GUI components (menu, buttons, keyboard hits – the *controller*). These are activated by the human user to change the graph and/or its drawing. Together with the GraphHandler structure (the *model*) they form a model-view-controller architecture [6].

The painting engine is sensitive to given node or link attributes. Node and link appearances (colour, sizes/widths, links directivity, etc.) can be changed through these attributes. This engine can also draw the part of the graph comprised in a given zone, with variable node and link size coefficients. It is thus possible to zoom over a portion of the graph. Nodes and links can be added, moved, or deleted in a drag and drop way. The GUI also offers a way to edit the attributes of each element.

Graph modifications can be automated by means of an embedded Groovy console (Groovy, which shares many similarities with Python, can be used as a scripting language for Java). In summary, the Javanco GUI permits one to handle a graph in much the same way that MATLAB permits one to handle vectors and matrixes.

Users' objects can add elements to the GUI in a simple way. Upon a right click on an element, the user object contained in the element (if any) is asked for menu items and corresponding actions. These menu items are added to the contextual menu that is popped-up.

### 2.3 XML import/export

Graphs can be imported and exported from and to various file formats in use in the graph and network community. Javanco also uses a specific XML format called MND [7]. Node and links are associated with XML elements, and their attributes stored as XML ones. As XML elements can contain sub-elements, these can be used by the user objects. In this way, more complex datasets such as routing tables or paths can be imported/exported from/to the same file. This makes the MND format very flexible.

### 2.4 Graph generators and other functions

Javanco embeds a collection of graph generators, that can be used to quickly generate simple regular structures (rings, grids, honey-nests) or more complex ones (random planar topologies, random trees, switch fabrics, etc.). Figure 1 displays some samples of the generated random planar graphs. These generators can be used as standalone objects (returning *GraphHandler* objects containing the desired graph when called) or through the GUI, via a dedicated interface.

As routing is omnipresent in networking experiments, Javanco includes a whole set of functionalities permitting one to compute one or all shortest paths in a graph (using the values associated with a given attribute such as link weights). Paths in a graph can also be enumerated using *breadth-first search* (*BFS*) algorithms. Returned *Path* objects can be split, combined, sorted, etc. Potential loops can also be detected or removed.

Finally, as Javanco is aimed at supporting Monte Carlo based simulations, it offers a collection of Pseudo Random Number Generators (PRNG) through a unified API. Experiments can hence be conducted using several recognized PRNGs (Mersenne Twister, Fourtap, etc.), a recommended practice [8].


*Figure 2. Graphical representation of an OBS simulation with deflection routing. A link cut is simulated by setting the number of channels to 0 (left). The impact of the cut on the other links can be observed (right)., Congestion is colour coded with red indicating high blocking.*

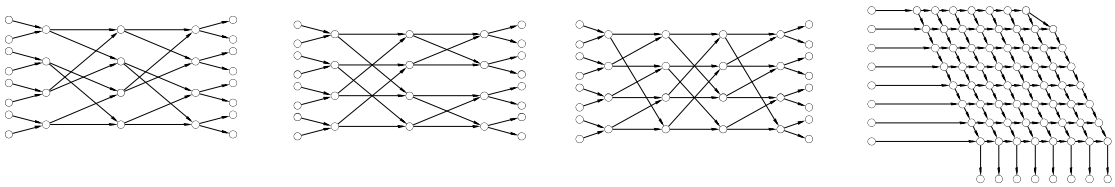

*Figure 3. Drawings of switch architectures generated in Javanco, allowing a visual analysis of the switch complexity*

## 3. GENERAL USAGE OF JAVANCO

With the features detailed so far, a user can create a graph manually, using the console or a generator. He can then save, reload and visualise it, and edit its characteristics.. In other words, the user can easily prepare the dataset that is then transmitted to an algorithm. As the algorithm terminates, the graph painting functions can be exploited to graphically display results. Link colours can then be employed to highlight the network bottlenecks, or new links can be added to represent an added logical topology. Taking advantage of the clear distinction between the model and the interface, Javanco can be used to display the state of the network during a simulation. The *user object* menu items previously mentioned may offer the user a way to change some network element characteristics, such as the capacity, the injected load, the routing, etc. User changes are incorporated *live* into the simulation, and the user can immediately see the effects as element colours reflect the updated states.

Javanco thus acts as a generic interface between an algorithm or a simulator and the user. It provides additional features as well. First, the graph generators offer a simple way to test an algorithm (and measure how well it achieves optimisation), over a wide set of graphs. Second, functionalities such as routing, clique detection, node proximity tests, etc. can be easily exploited by the algorithm, reducing in this way the algorithm development time. Third, the *GraphHandler* data structure can be used to bridge several algorithms without having to successively serialise the structure into a file and reload it and parse it. For example, it can introduce a virtual topology designed by a first tool into a generator placement algorithm.

### 3.1 Practical utilisations

Javanco has been used as substrate for several projects and experiments. The JAVOBS simulator [9] was built on top of it. Javanco's features are used at simulation initialisation, when the model is set-up and when the simulation objects are created according to the topology. The GUI can also be used to display the level of congestion experienced by links, as well as to set the number of OBS channels available on each link (Figure 2).

Regenerator placement algorithms have also been developed on top of Javanco [10-11]. In this context, the random graph generators have been extensively used to debug and validate the constraint programming based algorithms [11], by comparing their results with the ones obtained by enumeration methods [10]. Surprisingly, a difference between the two algorithms (due to an implementation bug) appeared only at the 120$^{th}$ generated graph. This shows how algorithm testing over a large number of automatically generated instances can be useful.

Javanco has been employed in several other studies [12-14]. Recently, we used it to design silicon photonics based switch fabrics (Figure 3), and simulate different packet based transmission schemes over them [15].

These experiences convinced us that having a framework able to carry out most graph-related operations is an asset. The GUI also revealed itself to be a major advantage: it shortens the learning curve for newcomers, allows interactive demos and eases reporting. It also helps the programmer verify the behaviour of the algorithm along development stages.

## 4. JAVANCO ARCHITECTURE COMPARED TO OMNET++

In contrast to other frameworks, Javanco core functionalities are limited to the graph level only. This helps maintain easy accessibility for new users. Additionally, implementing a simulation kernel directly into Javanco core classes would certainly make it more cumbersome for the experiments involving no simulation mechanisms. Algorithms involving simulation kernels must therefore define these separately (as was the case for JAVOBS). The implementation of these kernels is accelerated, however, by several structures and features available in the API (e.g. the event manager, *Time*, *Rate* and *DataSize* objects).

This modular approach contrasts with other simulation frameworks, such as OMNeT++. OMNeT++ simulations involve using the NED language and a lightweight configuration language in addition to a large API for passing messages between models and manipulating the event queue. While these features provide functionality for building network simulations, they also constrain the functionality of the software as a whole. For example, the configuration language included in OMNeT++ allows the user to specify batch simulations where parameters are varied over multiple simulation runs. However, the user has no control over how these parameters are combined, usually resulting in a brute force batch where every parameter combination is used. The lack of control in this configuration language makes it difficult to employ optimization algorithms that can save the user time, by reducing the number of simulations, and effort, by reducing the amount of simulation output data. OMNeT++ is mentioned here as an example of a simulation framework that, compared to Javanco, favours integrated functionalities at the price of reduced flexibility.

## 5. CONCLUSIONS

Javanco offers graph handling functionalities, together with a flexible GUI and a collection of routines and components often used in the context of optical network modelling. It has already been used to support several studies, and is currently being used and developed in our group to advance optical network modelling and design.

Javanco promotes a particular software development strategy: simulations or optimisation programs are *grown* over the substrate it provides, according to the requirement of each study. In this way, the user keeps control of his or her algorithm and delegates repetitive tasks to the framework. We believe that this software framework architecture is a valuable alternative to the more monolithic approaches proposed by other frameworks.

## REFERENCES


[1] S. Rumley *et al.* "*Software Tools and Methods for Research and Education in Optical Networks*." In Towards Digital Optical Networks, pp. 331-364. Springer, 2009.

[2] A. W. Bragg. "*Which network design tool is right for you?*" IT Professional vol 2(5) :23-32, 2000.

[3] J. Pan and R. Jain. "*A Survey of Network Simulation Tools: Current Status and Future Developments*", http://www1.cse.wustl.edu/~jain/cse567-08/ftp/simtools.pdf (Accessed April 2013), 2008.

[4] J. Sommer and J. Scharf. "*IKR Simulation Library*." In Modeling and Tools for Network Simulation, pp. 61-68. Springer, 2010.

[5] J. Chan *et al.* "*Phoenixsim: A simulator for physical-layer analysis of chip-scale photonic interconnection networks*." Conference on Design, Automation and Test in Europe, 2010.

[6] G. E. Krasner *et al.* "*A description of the model-view-controller user interface paradigm in the smalltalk-80 system.*" Journal of object oriented programming 1(3): 26-49, 1988.

[7] S. Rumley and C. Gaumier. "*Multilayer description of large scale communication networks*." In Recent Advances in Modeling and Simulation Tools for Communication Networks and Services. Springer, 2007.

[8] K. Pawlikowski *et al.* "*On credibility of simulation studies of telecommunication networks*." IEEE Communications Magazine 40(1):132-139, 2002.

[9] O. Pedrola *et al.* "*JAVOBS: a flexible simulator for OBS network architectures.*" Journal of Networks 5(2): 256-264, 2010.

[10] S. Rumley and C. Gaumier. "*Cost aware design of translucent WDM transport networks*." IEEE International Conference on Transparent Optical Networks (ICTON), 2009.

[11] S. Rumley *et al.* "*Multi-objective optimization of regenerator placement using constraint programming*." Conference on Optical Network Design and Modeling (ONDM), 2011.

[12] P. Pedroso *et al.* "*AnyTraffic routing algorithm for label-based forwarding.*" In Global Telecommunications Conference, (GLOBECOM) 2009.

[13] O. Pedrola *et al.* "*Modelling and performance evaluation of a translucent OBS network architecture*." In Global Telecommunications Conference (GLOBECOM), 2010.

[14] S. Rumley and C. Gaumier. "*Resource utilisation analysis in WDM backbone networks*" IEEE International Conference on Transparent Optical Networks (ICTON), 2011.

[15] G. Dongaonkar *et al.* "*Ultra-low Latency Optical Switching for Short Message Sizes in Cluster Scale Systems*", IEEE Optical Interconnects Conference, 2013.