OMNeT++ 6.0 and Beyond

András Varga
OMNeT++ Core Team

Contents

- OMNeT++ 6.0 Project Overview
- NED Improvements
- Statistics Recording Improvements
- OMNeT++ 6.0 Message Files
- Programming Model
  - Transmission Updates,
  - Pause Points
  - and more
- Simulation IDE / Qtenv
  - Analysis Tool
  - Sequence Chart Tool
  - Documentation Generator (neddoc)

The OMNeT++ 6.0 Project

In the making for 3-4 years

- Development started mid-2018, roughly after OMNeT++ 5.4 was released
- Rewriting of the Analysis Tool started in April 2017
- Release planned for Autumn 2021 (real soon!)

OMNeT++ 6.0

Multiple objectives

- Cleanup
  - Adjustments to NED and MSG files and some C++ APIs
  - Remove deprecated methods and classes, and obsolete code like Tkenv
  - C++ modernization
- Infrastructural improvements
  - E.g. new, versatile expression parser/evaluation framework under the hood
- Cater for INET's needs
  - structured (JSON) parameters, transmission updates, etc.
- Simulation IDE, Qtenv
  - Expressive, powerful Python-based Analysis Tool
  - Improved visualization in the Sequence Chart Tool
- Myriad of other improvements, adjustments, bug fixes

Expect more

This presentation only covers the most visible or practically most important changes, it is far from exhaustive.

Parameter References in NED

Parameter References in NED

Interpretation of parameter references within a submodule's or channel's curly brace block changed!

- A plain identifier (`foo`) no longer refers to the *compound module*'s parameter, but to the *local* parameter with that name!
- Use **`parent.`**`foo` to access the compound module's parameter!
- Similarly, a sibling submodule `bar` must now be referenced as **`parent.`**`bar`
- Outside subcomponent curly brace blocks, things are unchanged.

- Note: **`parent`** complements **`this`**, which already existed.

- *The only way to write NED code compatible with both OMNeT++ 4.x/5.x and 6.x is to use explicit qualifiers within curly brace blocks:* ***`this.`****`foo`*, ***`parent.`****`foo`*!

Example

```
module Node {
    parameters:
        int address;
        string appType;
     gates:
        inout port[];
    submodules:
        app: <appType> like IApp { ✓
            parameters:
                address = address;   ☠! → parent.address!
        }
        routing: Routing {
            gates:
                in[sizeof(port)];   ☠! → sizeof(parent.port)!
                out[sizeof(port)];   ☠! → sizeof(parent.port)!
        }
        queue[sizeof(port)]: L2Queue { ✓
        }
    connections:
        routing.localOut --> app.in; ✓
        routing.localIn <-- app.out; ✓
        for i=0..sizeof(port)-1 { ✓
            routing.out[i] --> { delay = routingDelay; }--> queue[i].in;  ☠! → parent.routingDelay!
            routing.in[i] <-- queue[i].out;  ✓
            queue[i].line <--> port[i]; ✓
        }
}
```

Motivation for the Change

- "`foo = foo;`" assignments in NED files were somewhat weird
- Context-dependent interpretation of expressions
  - Interpretation of an identifier depended on where it occurred (compound module scope or subcomponent scope)
  - A problem where an expression occurs out of context, or context is not obvious
    - in omnetpp.ini
    - in expression strings evaluated from C++ code

Porting, Compatibility

- Helpful error messages:

```
txRate = txRate;
```

<!>Error: Cannot assign parameter 'txRate': Parameter refers to itself in its assignment (did you mean 'parent.txRate'?) -- at Aloha.ned:25 -- in module (aloha::Host) Aloha.host[0] (id=3), during network setup

- Writing NED compatible with both 4.x/5.x and 6.x:
  Use explicit qualifiers within curly brace blocks: `this.foo`, `parent.foo`.

NED Support for Uniform Function Call Syntax

Uniform Function Call Syntax

What is Uniform Fuction Call Syntax (UFCS)?

- Alternative notation for function calls (“syntactic sugar”)
- Used in D and Nim, proposed for C++

$$f(x,y,...) \sim x.f(y,...)$$

- Examples in NED:
  - `substringAfter(“foobar”,”ba”) ~ “foobar”.substringAfter(“ba”)`
  - `startsWith(“foobar”,”foo”) ~ “foobar”.startsWith(“foo”)`
  - `length(“foo”) ~ “foo”.length()`
  - `replace(trim(”  foo “), “o”, ”x”) ~ ” foo ”.trim().replace(“o”, ”x”)`
  - `sin(0.5) ~ 0.5.sin()`

Allows chaining

Sometimes awkward
(use only where appropriate)

JSON Parameters

JSON - Motivation

- Structured parameters at last!

  Replaces other heterogenous solutions:
  - strings used as lists: “3 5 7 3 10 4 2”
  - XML
  - CSV files
  - custom files (like .rt files in older INET versions)

- But... what’s wrong with XML?
  - Too verbose:

    `<reallyLongTagName>...</reallyLongTagName>`
  - Attributes and child elements offer duplicate functionality:

    `<point><x>4</x><y>5</y></point>` versus `<point x=4 y=5/>`
  - Overkill for simple uses
  - Cannot (easily) be made parametric

JSON Parameters

Combination of two features:

- object parameters
  - `object packetToSend = createPacket();`
- JSON notation in expressions
  - `list = [ 1, 2, 3 ];`
  - `map = { “name”: “Joe”, “age”: 42 };`

Extensions to JSON:

- unquoted map keys (stored as string)
- expressions allowed anywhere (except in keys)
- tagged “objects” (dictionaries): `cMessage { “name”: “msg3” };`

Ini File JSON Examples

Example parameter assignments:

```
# plain list
*.switch.gptp.masterPorts = ["eth1", "eth2"]

# list of maps
*.switch2.macTable.addressTable = [{address: "switch1", interface: "eth0"},
                                   {address: "sink1", interface: "eth1"},
                                   {address: "sink2", interface: "eth2"}]

# tagged object
*.app.msgToSend = omnetpp::cMessage { name : "hello", kind: 7 }

# units, expressions, parameter references accepted
*.app.traffic = {
                   { length : 100B, interval: 10ms },
                   { length : defaultPkLen, interval: 1.5 * defaultInterval }
                ]
```

Typed Object Example

Trail.msg file defines data classes:

```
struct Point {
    double x;
    double y;
}

class Trail extends cOwnedObject {
    Point points[];
}
```

NED (or ini): Use “tagged object” syntax to fill a Trail object directly:

```
object trail = Trail {
  points: [
    { x: 1, y : 5 },
    { x: 2, y : 6 },
    { x: 3, y : 7 },
    { x: 4, y : 8 }
  ]
};
```

Object Ownership

- How ownership is handled:
  - Module parameters want to exclusively own the objects assigned to them
  - `fooCopy = foo;  // error`
  - `fooCopy = dup(this.foo); // ok`

C++ API

elem → `cValue`

- variant of (bool, intval_t+unit, double+unit, string, pointer/object)
- getType(), boolValue(), intValue(), doubleValue(), stringValue(), pointerValue(), objectValue(), doubleValueInUnit(), doubleValueRaw() + getUnit(), ...

[...] → `cValueArray`

- contains cValue objects (which may hold a primitive value, or further cValueArray, cValueMap or arbitrary other objects)
- internally: std::vector<cValue>
- add(), insert(), get(), set(), erase(), operator[], getArray() [→ std::vector], ...

{...} → `cValueMap`

- internally: std::map<std::string,cValue>
- set(), get(), containsKey(), erase(), operator[], getFields() [→ std::map], etc.

Example C++ Code

Example 1:

```
**.foo.bar.addressTable = [{address: "switch1", interface: "eth0"},
                           {address: "sink1", interface: "eth1"},
                           {address: "sink2", interface: "eth2"}]

// C++:
cValueArray *addressTable = check_and_cast<cValueArray *>(par("addressTable").objectValue());
for (int i = 0; i < addressTable->size(); i++) {
       cValueMap *entry = check_and_cast<cValueMap *>(addressTable->get(i).objectValue());
       std::string address = entry->get("address").stringValue();
       std::string interface = entry->get("interface").stringValue();
       table[address] = interface;
}
```

Example 2:

```
**.foo.bar.trail = Trail {
      points: [
        { x: 1, y : 5 },
        { x: 2, y : 6 },
        { x: 3, y : 7 },
        { x: 4, y : 8 }
      ]
    };

// C++:
Trail *trail = check_and_cast<Tail *>(par("trail").objectValue());
```

Statistics Recording Improvements

Demultiplexing Statistics: Motivation

Example: Received packet statistics in some app/protocol

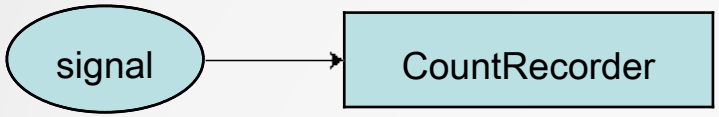
```
simple SomeUdpApp {
  parameters:
    @signal[packetReceived](type=cPacket);
    @statistic[packetReceived](source=packetReceived; record=count, ...);
    ...
}
```

Problem:

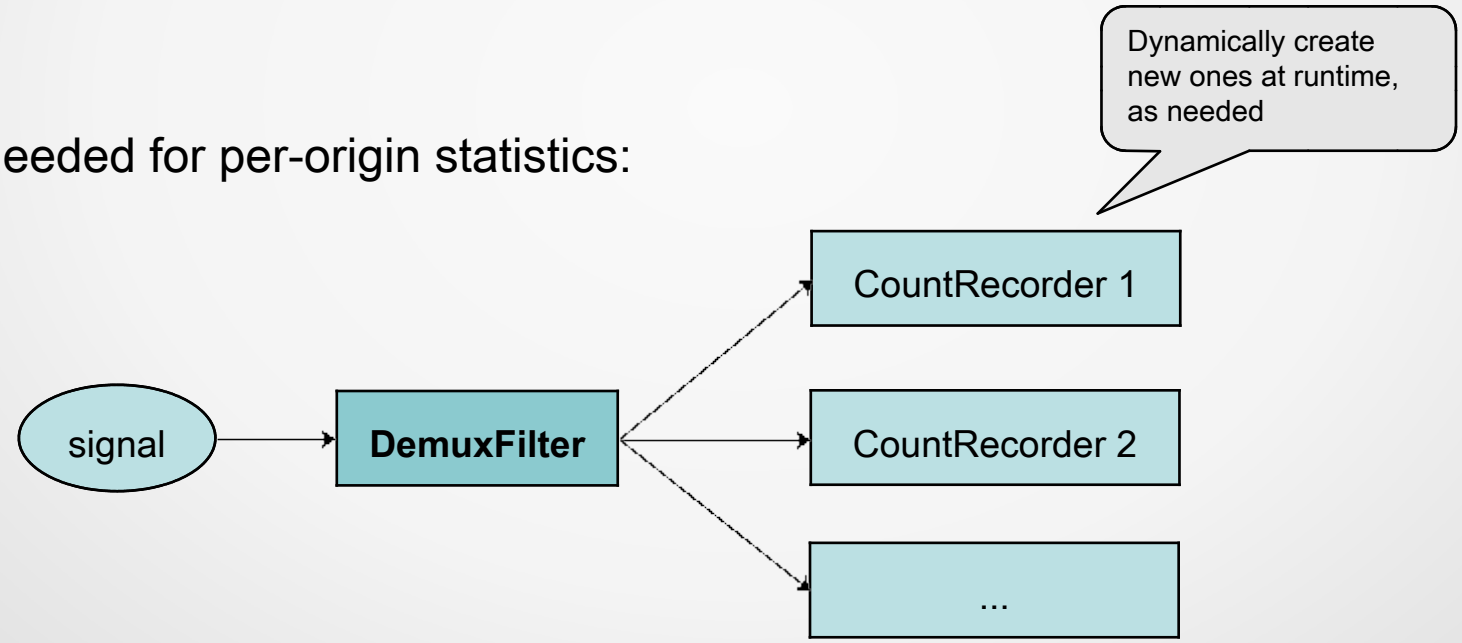
- One app records a single packetReceived:count scalar
- How to record received packet statistics per origin? (by retaining `@statistic` usage)

Conceptual Solution

Original statistic:



Needed for per-origin statistics:



Demux Filter

Updated module:

```
simple SomeUdpApp {
   parameters:
     @signal[packetReceived](type=cPacket);
     @statistic[packetReceived](source=demux(packetReceived); record=count, ...);
     ...
}
```

How the demux filter works:

- Input for classification is the name string of the “details” object emitted with the value

```
emit(<signal> ,<value>, cObject *details = nullptr);
```

- Demux filter clones the rest of the chain for each distinct name

Code example:

```
cNamedObject details(pk->getSourceAddress().str().c_str());
emit(pkReceivedSignal, pk, &details);
```

Records one scalar per each sender:

- `"packetReceived:count:10.0.0.1"`
- `"packetReceived:count:10.0.0.2"`
- etc.

Updates to Warmup-Period Handling

What is warmup period?

- a way to remove the effect of the initial transient from the statistics

Default warmup period handling:

- If the `warmup-period=<duration>` config option is present, a WarmupPeriodFilter is inserted at the **front** of the filter/recorder chain of every statistic.
- WarmupPeriodFilter is like a timed switch: discards values during the warmup period, and lets them through afterwards

Issue: "front" is not always the correct place for the wamup period filter!

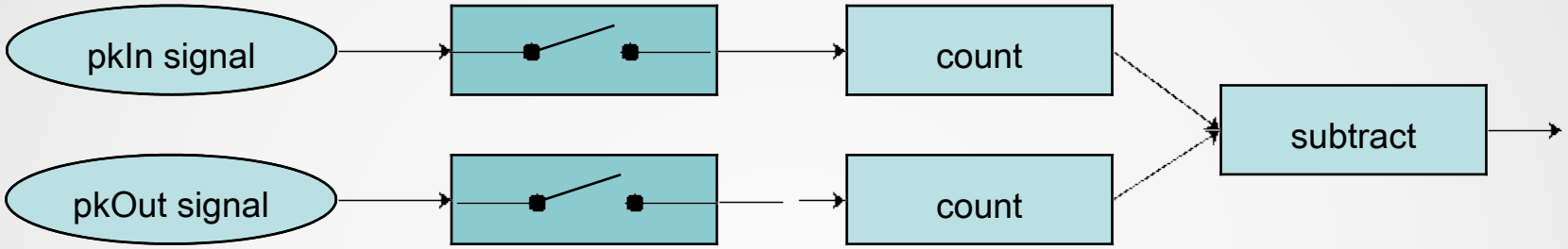
Consider computing number of packets in a (compund) queue as #arrivals - #departures:

```
@signal[pkIn](type=cPacket);
@signal[pkOut](type=cPacket);
@statistic[queueLen](source=count(pkIn)-count(pkOut);record=vector);
```

- What happens when we configure a warmup period?

Illustration of the Issue

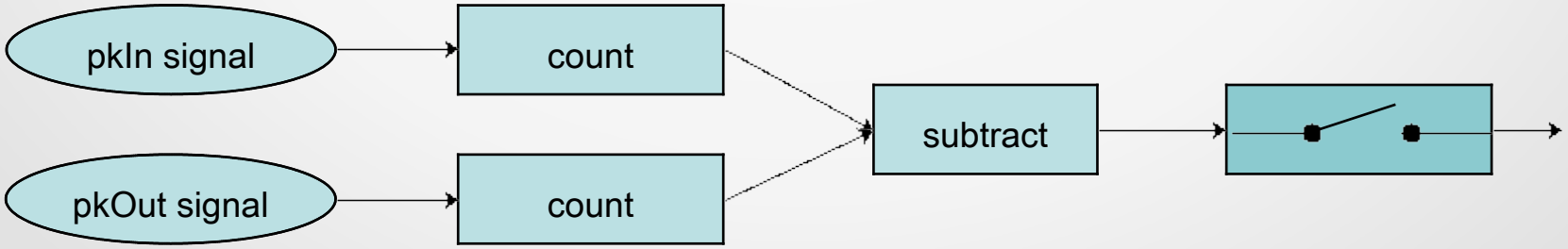
Default warmup period filter placement:



corresponds to: `count(warmup(pkIn))-count(warmup(pkOut))`

NOTE: At the end of the warmup period, result is zero instead of the actual number of packets already in the queue (then it may also go negative)!

Rather, we need:



`warmup(count(pkIn)-count(pkOut))`

Warmup Period: Solution

Made it possible to:

1. disable automatic adding of the warmup filter
2. add it manually at the right place

The updated statistic:

```
@statistic[queueLen](source=warmup(count(pkIn)-count(pkOut));
                     autoWarmupFilter=false;
                     record=vector);
```

Message Files in OMNeT++ 6.0

History

- OMNeT++ 5.3:
  - Message compiler internals reimplemented, language extensively revised
  - Message compiler contained both implementations: old one used by default, new one could be turned on with **--msg6**.
  - The **--msg4** option was also accepted as a no-op
  - INET started using **--msg6** right away (from INET 4.0)

- OMNeT++ 6.0:
  - **--msg6** mode became the default (**--msg4** code removed)
  - OMNeT++ 5.x (x>=3) recommended for porting msg files!

- Two main points:
  - support for imports
  - extensive configurability of generated code via properties

Imports

In 4.x mode, message files were processed in isolation

- For one msg file to use classes defined in another, you'd have to manually announce them

From 6.0, this is done via imports

- `import foo.bar.foobar` means:
  - find a `foo/bar/foobar.msg` file under one of the MSG *import path* directories, and
  - load the declarations from it
- Import path can be specified to **opp_msgtool** with **-I**<dir>
- Simplifies writing dependent msg files

Import Example

Before:

```
cplusplus {{
#include "inet/linklayer/common/MACAddress.h"
#include "inet/linklayer/ieee80211/mac/Ieee80211Frame_m.h"
}}

class noncobject MACAddress;
packet Ieee80211Frame;

packet Ieee80211MgmtFrame extends Ieee80211Frame {
  ...
}
```

After:

```
import inet.linklayer.common.MACAddress;
import inet.linklayer.ieee80211.mac.Ieee80211Frame;

packet Ieee80211MgmtFrame extends Ieee80211Frame {
  ...
}
```

Other Improvements

Changes:

- cObject is no longer the default base class!
- Field getters now return const reference; use get<name>ForUpdate() methods if you need to modify the returned object
- Inserter and eraser methods added for dynamic arrays.

Customization possibilities:

- Method names and various traits of the class or fields can be controlled via field properties
- Support for pointer and owned pointer fields (@owned field property)
- Custom fields: no code is generated, it can be added via cplusplus blocks (@custom)
- Support for pass-by-value for fields (@byValue)
- Inject code via targeted cplusplus blocks*
- ...

- Use **opp_msgtool -h builtindefs** to see the list of supported class/field properties

Targeted C++ Blocks

Basic syntax:

- `cplusplus {{ ... }}` – content is inserted into generated header file

Extended syntax:

- `cplusplus(target) {{...}}` – content is inserted where target specifies

Targets:

- `cplusplus(cc)`: into generated implementation (*_m.cc) file
- `cplusplus(Hello)`: into the class declaration of “Hello”
- `cplusplus(<methodname>)`: into the specified method, e.g.:
  - `cplusplus(Hello::Hello)`
  - `cplusplus(Hello::Hello&)`
  - `cplusplus(Hello::~Hello)`
  - `cplusplus(Hello::operator=)`
  - `cplusplus(Hello::setFoo)`
  - `ccplusplus(Hello::getFoo)`

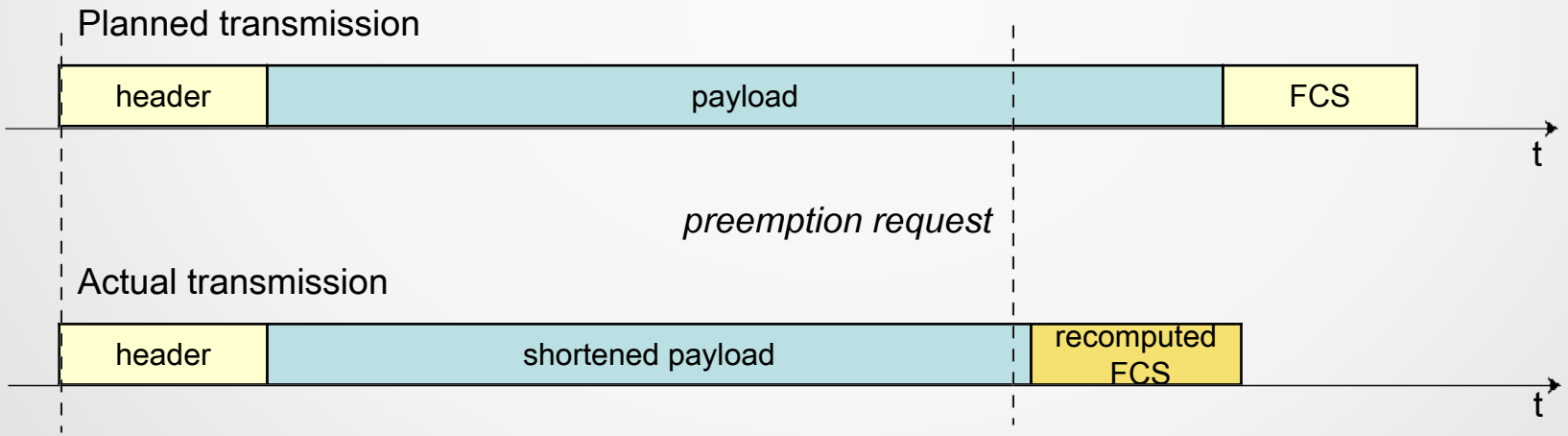
Programming Model

Transmission Updates

- **Motivation**: Standardize signalling end of transmission (e.g. frame preemption or abortion) on wired/wireless links
  - Normal end-of-tx is usually implicit
  - Various ad-hoc solutions were in use for signalling tx abortion:
    - forceTransmissionFinishTime() hack for wired
    - special messages/packets
    - method calls
    - combinations of the above
  - ... done outside the knowledge of the simulation kernel!
  - Therefore, correct visualization in Qtenv and Sequence Chart was impossible

Example: Ethernet Frame Preemption

- Two frame priority levels: *normal*, *express*
- An express frame can preempt transmission of a normal frame
- Normal frame needs to be finished:
  - at least the current octet finished
  - updated FCS transmitted



Transmission Updates

Solution:

- A specially marked packet is sent, which carries:
  1. updated packet content
  2. remaining transmission duration
- Normal end-of-tx can still be implicit

C++ API:

- `SendOptions` class added
- All send methods received an extra `const SendOptions& options` argument
- `SendOptions` carries:
  - info that packet is a transmission update
  - reference to original packet (transmissionId)
  - remaining duration (which may be 0)
- `SendOptions` may also carry other info, unrelated to tx updates

Pause Points

- Motivation: Proliferation of synchronous method calls among modules in INET 4.x
  - E.g. module A calls into module B, module B calls into module C, module C sends a packet →
  - Single simulation event, multiple modules involved!
  - Hard to see in the Qtenv GUI what happens

Pause Points, cont'd

Idea: Add *pausePoint()* calls into modules at strategic places

1. pausePoint() calls into Qtenv, which (optionally) starts a new GUI event loop
2. This allows user to interact with the simulation *while inside the pausePoint() call*
3. Resuming causes Qtenv to quit inner GUI event loop and return from pausePoint() call, simulation continues as if nothing happened

Implementation is in progress.

More...

- An incomplete list of simulation kernel changes that affect how models are programmed:
  - scheduleAt() complemented with scheduleAfter(), rescheduleAt(), rescheduleAfter()
  - Added preDelete() virtual method
  - Added component_ptr (module/channel weak pointer)
  - Type safety in cClassDescriptor via any_ptr (new class)
  - Subtle changes in: parameter change notification, initialization, destruction, submodule array representation
  - String utility functions made accessible
  - More powerful cStringTokenizer
  - More powerful expression evaluator
  - ....

Simulation IDE, Qtenv

Qtenv

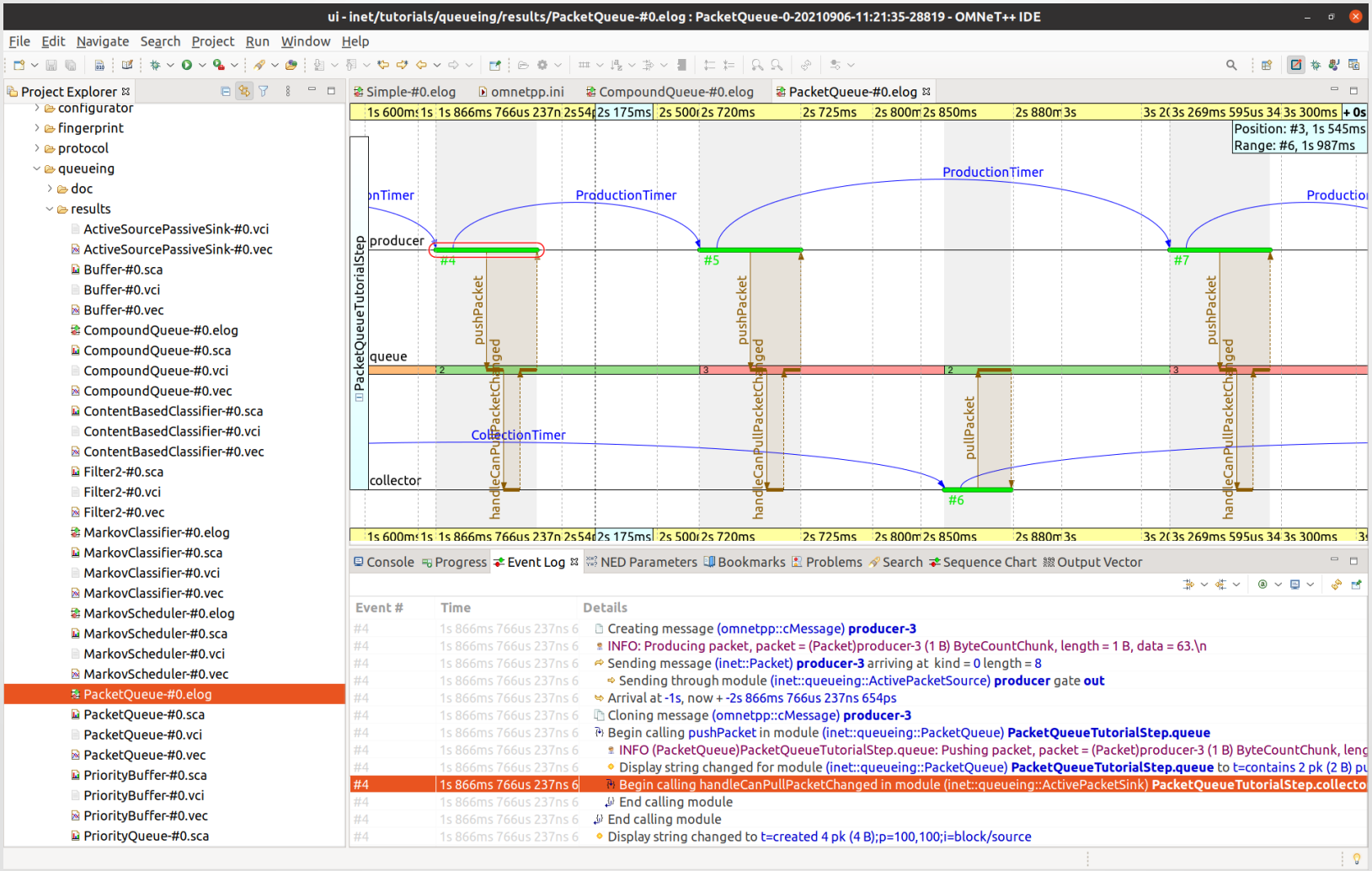
- Modern look
  - New icons, dark mode, HIDPI mode
- Features
  - Layout groups (unrelated modules may be arranged in row/matrix/etc.)
  - Submodule display names (useful for assigning descriptive names to submodule vector elements like “app[0]”, “app[1]”, etc.)
  - Visualization of transmission updates
  - Pause Point support
- Improvements
  - Improved, more featureful message/packet traffic view in the log inspector (digit grouping, etc.)
  - Object inspector improvements
  - Many more

Sequence Chart Tool

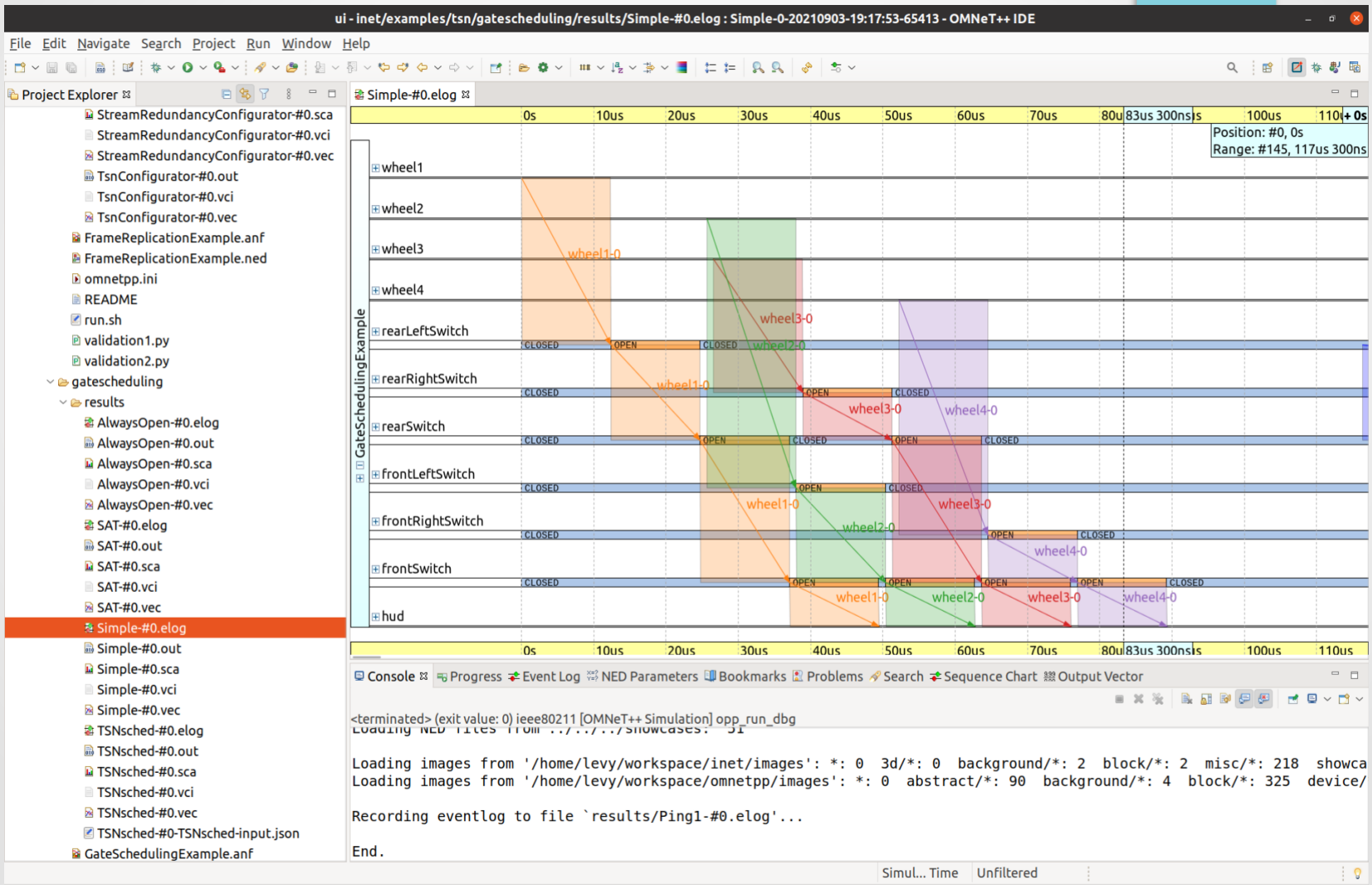
Several improvements:

- Horizontally expanded events, for better visualization of nested method calls
- User-defined coloring of events, message sends, method calls, axes, etc.
- Foldable compound module axes
- Set of axes to be displayed can be chosen
- Improved scalability and reliability

SeqChart: Method Calls



SeqChart: Coloring



Analysis Tool

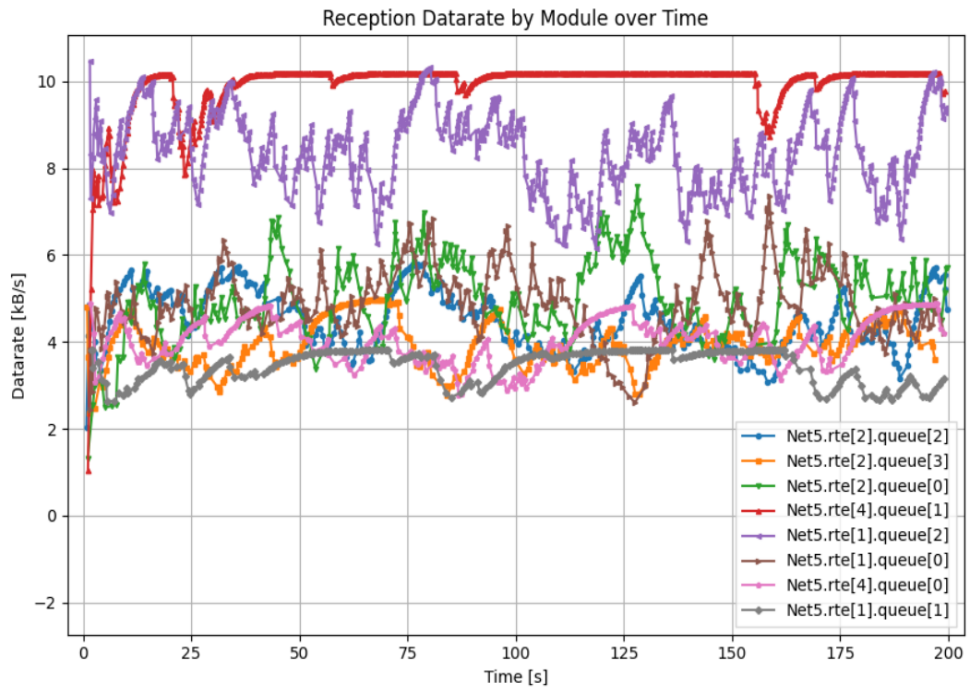
- Superficially similar to previous version
  - *Inputs* page, *Browse Data* page, *Charts* page (NEW!), *Chart* pages
  - However, it allows *arbitrary computations* or processing, offers *unlimited possibilities for visualization*, *publication quality output*, and more.
- Internally: Charts driven by Python scripts
  - Utilizes: **Pandas, NumPy, SciPy, MatplotLib**
  - Rendering via Matplotlib (flexibility, features) or native plotting widgets (performance)
  - Chart configuration edited via dialogs
  - Scripts (and dialogs) fully editable
  - Easy to add new chart types (just export existing chart as “chart template”)
  - **opp_charttool** allows charts can be run on the command line too, for viewing or image/data export
- Previous presentations:
  - At the 5th and 6th OMNeT++ Community Summits (2018 Pisa, Italy; 2019 Hamburg, Germany), by Attila Török

Obligatory Screenshot

Fifo1.anf | Net10Experiment.anf | *Net5SaturatedQueue.anf

Reception datarate

```
import math
from omnetpp.scave import results, chart, utils, plot, vectorops as ops

# get chart properties
props = chart.get_properties()
utils.preconfigure_plot(props)

# collect parameters for query
filter_expression = props["filter"]
start_time = float(props["vector_start_time"] or -math.inf)
end_time = float(props["vector_end_time"] or math.inf)

# query vector data into a data frame
try:
    df = results.get_vectors(filter_expression, include_attrs=True,
                             include_itervars=True, start_time=start_time
except ValueError as e:
    plot.set_warning("Error while querying results: " + str(e))
    exit(1)

if df.empty:
    plot.set_warning("The result filter returned no data.")
    exit(1)

# apply vector operations
df = utils.perform_vector_ops(df, props["vector_operations"])

# plot
utils.plot_vectors(df, props)

utils.postconfigure_plot(props)

utils.export_image_if_needed(props)
utils.export_data_if_needed(df, props)
```



Inputs | Browse Data | Charts | Reception datarate

Problems | Module Hierarchy | NED Parameters | NED Inheritance | Console

Chart script output of Reception datarate

Analysis API

- Chart scripts usually begin with:
  - `from omnetpp.scave import results, chart, plot, utils, vectorops as ops`
- Terminology:
  - "chart" is what you edit (Python script + configuration)
  - "plot" is the artifact created by running the "chart"
- The packages:
  - **`results`**: Querying results into Pandas data frames
  - **`chart`**: Access to chart properties
  - **`plot`**: Plot to the IDE native plot widget
  - **`utils`**: Common interface to MatplotLib and native widgets; misc utility functions
  - **`vectorops`**: Vector operations (window average, running sum, etc)
  - **`analysis`**: Read/write/run ANF files from standalone scripts

Example Chart Script

```
# barchart.py (abbreviated)

import math
import numpy as np
import pandas as pd
from omnetpp.scave import results, chart, plot, utils

# get chart properties
props = chart.get_properties()
utils.preconfigure_plot(props)

# collect parameters for query
filter_expression = props["filter"]

# query scalar data into dataframe
try:
    df = results.get_scalars(filter_expression,
            include_attrs=True, include_itervars=True,
            include_runattrs=True)
except ValueError as e:
    plot.set_warning("Error while querying results: " + str(e))
    exit(1)

if df.empty:
    plot.set_warning("The result filter returned no data.")
    exit(1)

# determine "groups" and "series" for pivoting
groups = utils.split(props["groups"])
series = utils.split(props["series"])

[omitted: checking and defaults for groups/series]

# names for title
names = ", ".join(utils.get_names_for_title(df, props))

# pivoting and plotting
df.sort_values(by=groups+series, axis='index', inplace=True)
df = pd.pivot_table(df,
            index=groups, columns=series,values='value')
utils.plot_bars(df, props, names)

# postconfigure, perform requested exports
utils.postconfigure_plot(props)

utils.export_image_if_needed(props)
utils.export_data_if_needed(df, props)
```

Documentation Tool

- Generates online documentation from NED
  - e.g. INET Framework NED Reference
- Now accessible from the command line
  - In 4.x and 5.x it was only available from the IDE
  - New command-line utility: **opp_neddoc**
    (runs the IDE in headless mode)
- Pages may contain externally generated content
  - Extra tables, etc.
  - Content specified via XML
  - Use: Documenting observed behavior in INET (sent/received packet tags, cross-module method calls; collected during running the example simulations)

Questions?