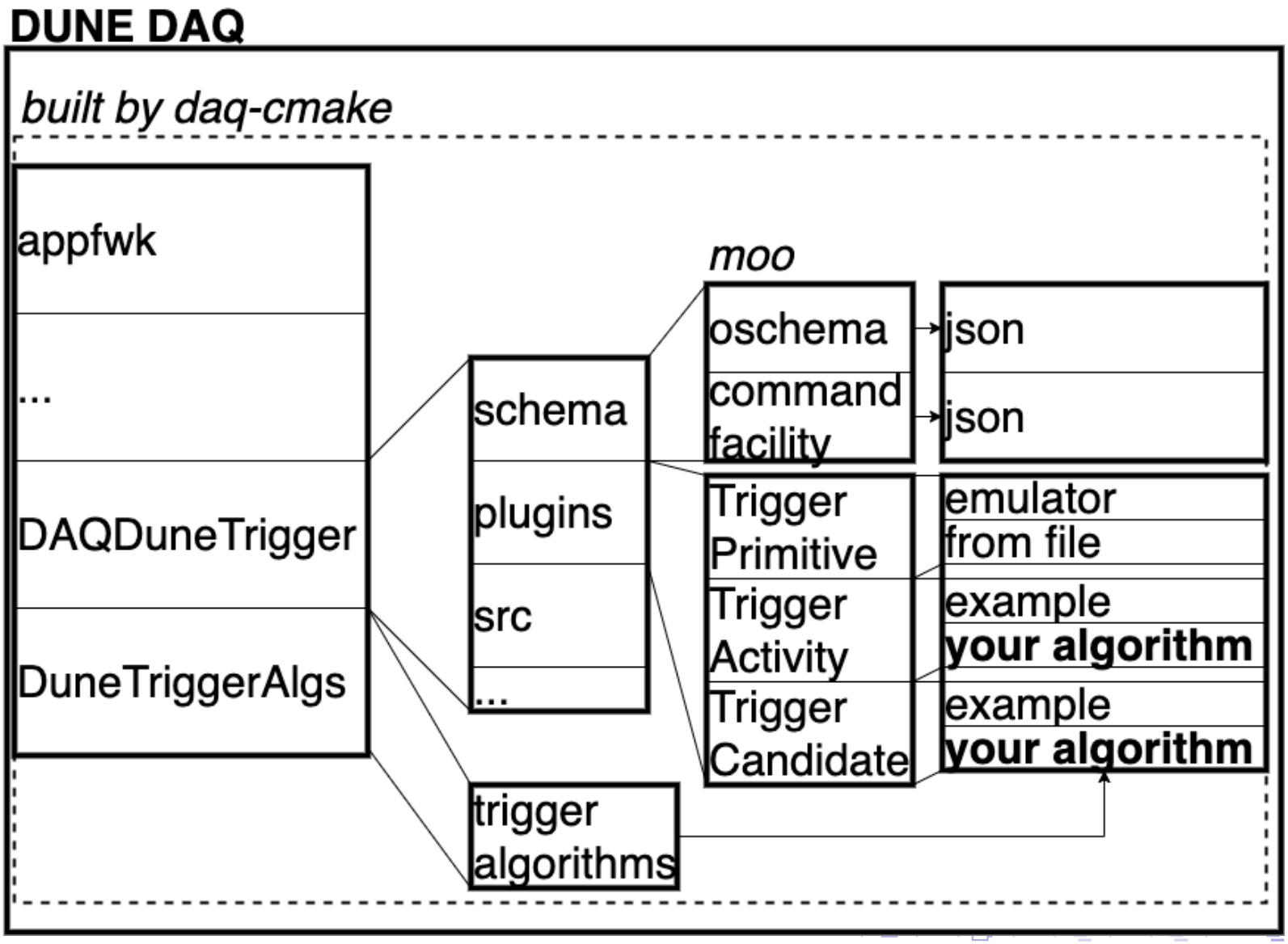
Data Selection Tutorial

Lukas Arnold, Thiago Bezerra

Columbia University, University of Sussex

February 9, 2021

Concept


DUNE DAQ
built by daq-cmake
appfwk
...
DAQDuneTrigger
DuneTriggerAlgs
schema
plugins
src
...
trigger algorithms
moo
oschema
command facility
Trigger Primitive
Trigger Activity
Trigger Candidate
json
json
emulator
from file
example
your algorithm
example
your algorithm

Setup environment

See

- Documentation: https://github.com/DUNE-DAQ/appfwk/wiki/Compiling-and-running-under-v2.2.0
- Tutorial from Carlos Barajas: https://indico.fnal.gov/event/47545/contributions/207745/attachments/139624/175288/Minidaq_app_Tutorial_5.pdf

Let's define basic parameters:

- DUNE DAQ version: export VERSION=dunedaq-v2.2.0
- Working directory: export MYDIR=dunedaq

Setup environment

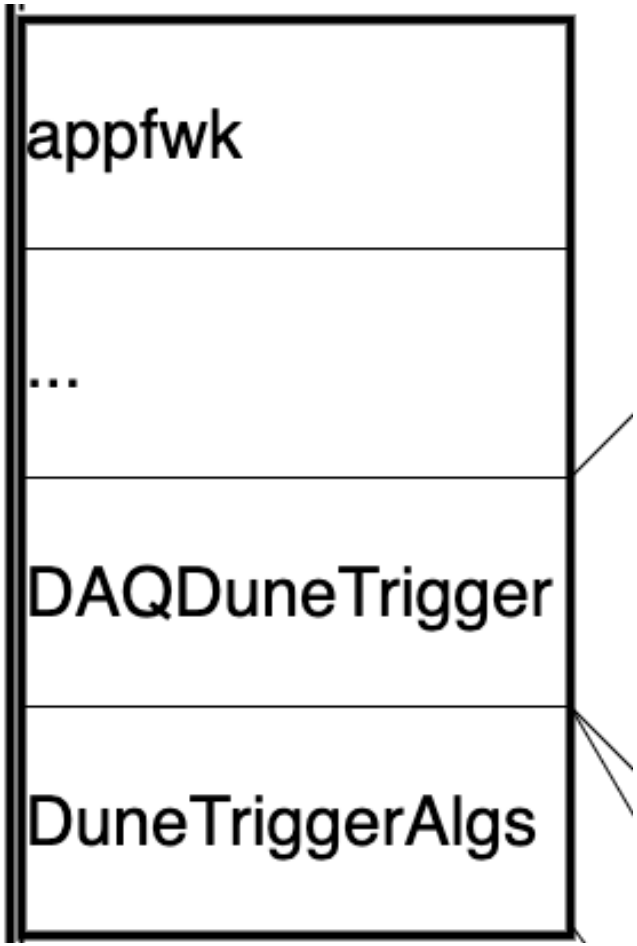
Setup daq-buildtools

$ git clone https://github.com/DUNE-DAQ/daq-buildtools.git -b $VERSION
$ source daq-buildtools/dbt-setup-env.sh
Expected answer: *Added /your/path/to/daq-buildtools/bin to PATH*
Added /your/path/to/daq-buildtools/scripts to PATH
DBT setuptools loaded

Setup working directory

$ mkdir $MYDIR
$ cd $MYDIR
$ dbt-create.sh $VERSION
This may take a couple of minutes and will create your working area in the $MYDIR location.

Setup environment



DUNE DAQ consists of several modules. The most important one is `appfwk` that provides the basic functionalities.
The data selection is based on two modules: `DAQDuneTrigger` and `DuneTriggerAlgs`.

Setup environment

The source code for the modules are located in `sourcecode`*:*

$ cd sourcecode

Now we use a couple of modules:

$ MODULES="appfwk DAQDuneTriggers DuneTriggerAlgs cmdlib ers filecmd listrev restcmd"

Let's check out each of these. Please note that $VERSION *is checked out only when it exists, and the DS repos are not part of* `DUNE-DAQ` *yet.*

$ for MODULE in $MODULES

> do; git clone https://github.com/DUNE-DAQ/$MODULE.git

> [[-d $MODULE]] || git clone https://github.com/thiagojcb/$MODULE.git

> cd $MODULE

> git fetch;git checkout $VERSION;git checkout he;git checkout moo

> cd ..; done

You should now have a complete working environment!

Example

In this tutorial, we will use `$MYDIR/sourcecode/DAQDuneTrigger/Plugins/TriggerPrimitiveFromFile.cpp` as an example.
It simply takes a (shorter) waveform from a `csv` file indicated by the user, formatted as follows:

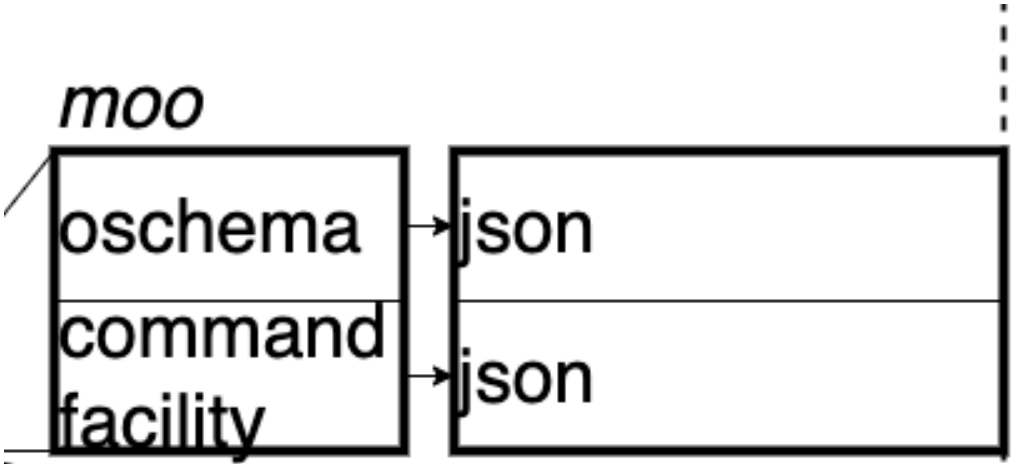
Format

time_start	time_over_threshold	time_peak	channel	adc_integral	adc_peak	detid	type
<value>,	<value>,	<value>,	<value>,	<value>,	<value>,	<value>,	<value>
...							
...							

moo

We want our module and our plugins to be **configurable**. This can be done using either `python` or `jsonnet` using `moo`. There are three different `moo`-based file:

- the plugin schema
- the plugin make
- the app / command facility



Documentation:
https://brettviren.github.io/moo/moo.html

Schema file

schema/test_schema.jsonnet

```
local types = {
    pathname : s.string("Path", "path",
doc="File path, file name"),

    conf: s.record("Conf", [
        s.field("filename", self.pathname,
"/tmp/example.csv",
                doc="File name for trigger primitives"),
    ], doc="TriggerPrimitiveFromFile configuration"),
};
moo.oschema.sort_select(types, ns)
```

Calling schema objects

plugins/test_plugin.cpp

```
...
void TriggerPrimitiveFromFile::do_configure(
const nlohmann::json& config /*args*/)
{
 auto params = config.get<triggerprimitivefromfile::Conf>();
 filename = params.filename;
}
...
```

Compiling the schema

in $MYDIR/sourcecode/DAQDuneTriggers

Should be done in `build` *process at some point.*

```
$ moo -g '/lang:ocpp.jsonnet' -M schema -A
path=dunedaq.DAQDuneTrigger.triggerprimitivefromfile -A
ctxpath=dunedaq -A
os=DAQDuneTriggers-TriggerPrimitiveFromFile-schema.jsonnet render
omodel.jsonnet onljs.hpp.j2
$ moo -g '/lang:ocpp.jsonnet' -M schema -A
path=dunedaq.DAQDuneTrigger.triggerprimitivefromfile -A
ctxpath=dunedaq -A
os=DAQDuneTriggers-TriggerPrimitiveFromFile-schema.jsonnet render
omodel.jsonnet ostructs.hpp.j2
```

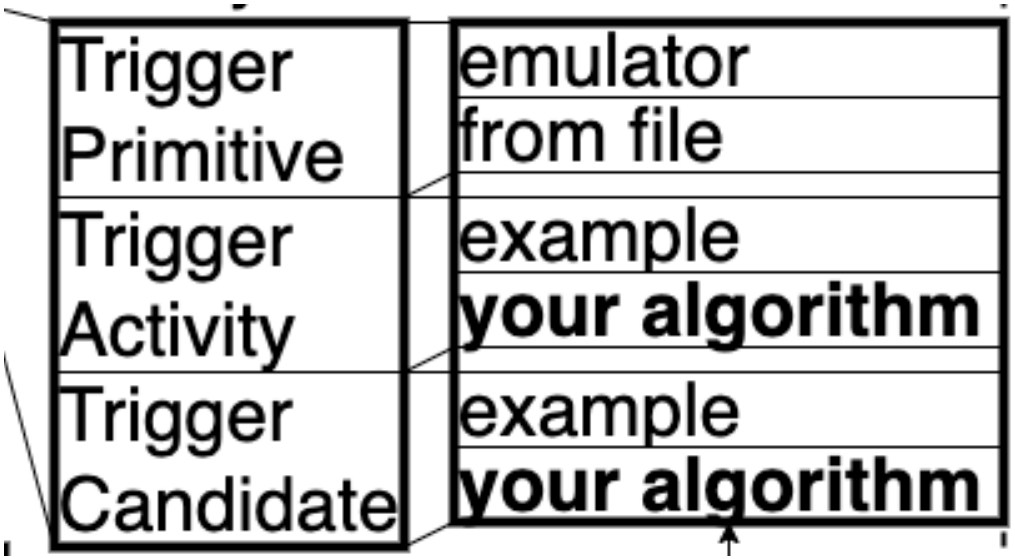
Include the `Nljs.hpp` *file in the plugin source code!*

Create the make

plugins/test_make.jsonnet

```
{
    conf(filename) :: {
        filename: filename,
    },
}
```

Creating the app



Creating the app

Includes

```
local moo = import "moo.jsonnet";
local cmd = import "appfwk-cmd-make.jsonnet";
local TPsGenerator = import "test_make.jsonnet";
```

Creating the app

Queues

```
local queues = {
        TPsQueue: cmd.qspec("TPsQueue",
                "FollyMPMCQueue",
                1000),

        TAsQueue: cmd.qspec("TAsQueue",
                "FollyMPMCQueue",
                100),

        TCsQueue: cmd.qspec("TCsQueue",
                "FollyMPMCQueue",
                10),
}
```

Creating the app

Plugins

...

```
        TPsGenerator: cmd.mspec("TPsGenerator2",
                "TriggerPrimitiveFromFile",
                [cmd.qinfo("output",
                        "TPsQueue",
                        cmd.qdir.output)]),

        TAsGenerator: cmd.mspec("TAsGenerator",
                "DAQTriggerActivityMaker",
                [cmd.qinfo("input",
                        "TPsQueue",
                        cmd.qdir.input),
                cmd.qinfo("output",
                        "TAsQueue",
                        cmd.qdir.output)]),
```

Creating the app

Configuration

```
[
        cmd.init([      queues.TPsQueue,
                        queues.TAsQueue,
                        queues.TCsQueue],
                [       modules.TPsGenerator,
                        modules.TAsGenerator,
                        modules.TCsGenerator])
                { waitms: 1000},
        cmd.start(40){ waitms: 1000},
        cmd.stop(){ waitms: 1000},
...
```

Creating the app

Configuration

```
...
        cmd.conf(
                [cmd.mcmd("TPsGenerator3",
                    TPsGenerator.conf("/tmp/csv_file.csv")),
                cmd.mcmd("DAQTriggerActivityMaker"),
                cmd.mcmd("DAQTriggerCandidateMaker"),)]
]
```

Creating the app

Compiling the command facility

Let's compile the app to provide a command facility

$ moo compile test_app.jsonnet > test_compiled.json

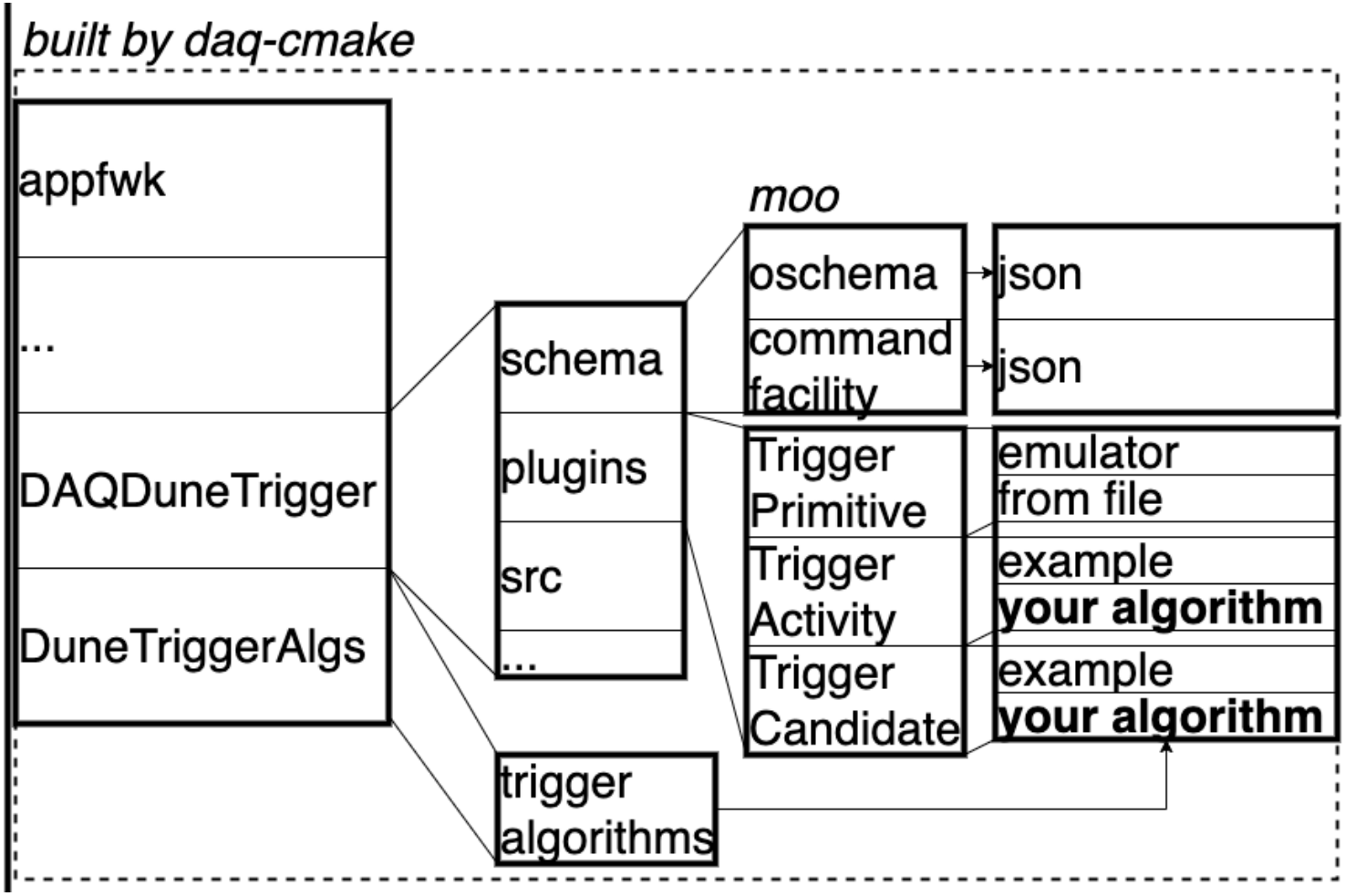
Creating the app

Compiling the module

Each module has a `CMakeLists.txt` file for build instructions.

```
cmake_minimum_required(VERSION 3.12)
project(DAQDuneTriggers VERSION 2.0.0)
find_package(daq-cmake REQUIRED )
daq_setup_environment()
find_package(appfwk REQUIRED)
find_package(DuneTriggerAlgs REQUIRED)
daq_add_plugin(TriggerPrimitiveRadiological \
     duneDAQModule LINK_LIBRARIES appfwk::appfwk)
daq_add_plugin(TriggerPrimitiveFromFile     \
    duneDAQModule SCHEMA LINK_LIBRARIES appfwk::appfwk)
...
```

Building DUNE DAQ



Building DUNE DAQ app

Let's build the DAQ App! $ dbt-setup-build-environment
$ dbt-build.sh –install

Running!

Let's run!

$ dbt-setup-run-environment

$ daq_application -c test_compiled.json

Now the commands `init | conf | start | stop` *should appear. Let's run the commands in this order to test the commands!*

Plugins

Introduction to the algorithms/plugins of `DAQDuneTriggers` and `DuneTriggerAlgs`.

What each module is in charge of?

- DuneTriggerAlgs
  - Define trigger "objects"
    - Trigger Primitive
    - Trigger Activity
    - Trigger Candidate
    - Trigger Decision (not used for now)
  - Define algorithms
    - Maker (finder) for each of the object
- DAQDuneTriggers
  - Each algorithm is it's own DAQProcess
  - Defines queue, etc

Data structure (TP, TA, TC) on DuneTriggerAlgs

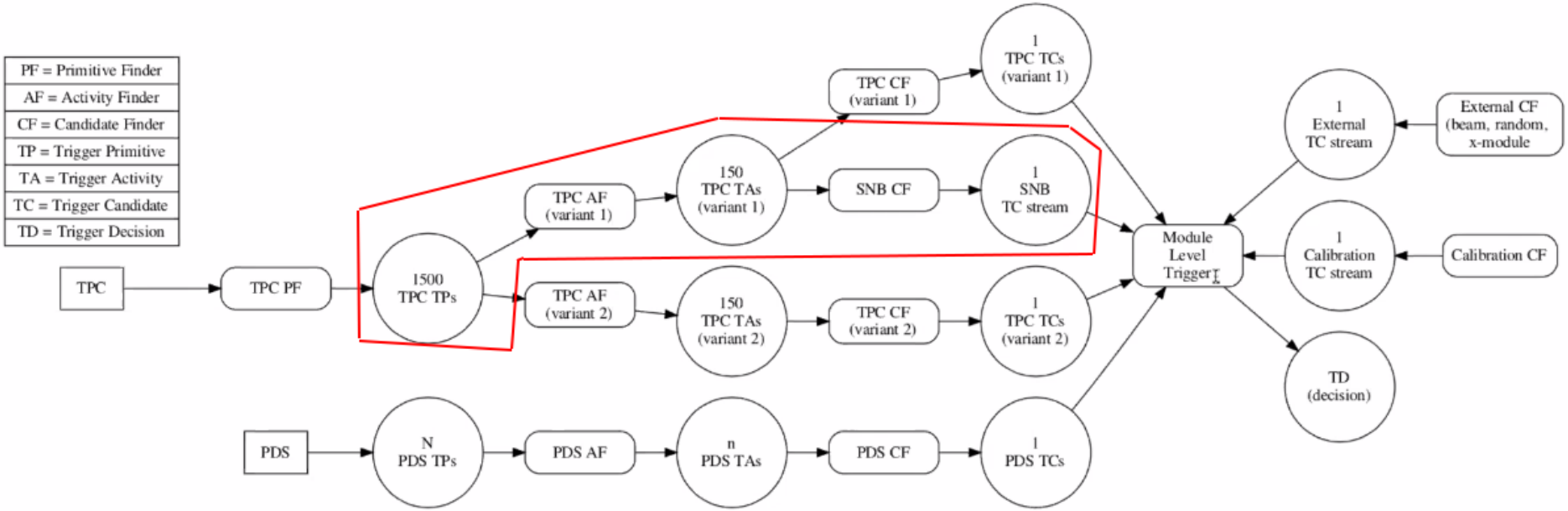
```
namespace DuneTriggerAlgs {
  struct TriggerPrimitive {
    int64_t  time_start          = {0};
    int64_t  time_peak           = {0};
    int32_t  time_over_threshold = {0};
    uint32_t channel             = {0};
    uint16_t adc_integral        = {0};
    uint16_t adc_peak            = {0};
    uint32_t detid               = {0};
    uint32_t type                = {0};
    uint32_t algorithm           = {0};
    uint16_t version             = {0};
    uint32_t flag                = {0};
  };
```

```
namespace DuneTriggerAlgs {
  struct TriggerActivity {
    int64_t  time_start    = {0};
    int64_t  time_end      = {0};
    int64_t  time_peak     = {0};
    int64_t  time_formed   = {0};
    uint32_t channel_start = {0};
    uint32_t channel_end   = {0};
    uint32_t channel_peak  = {0};
    uint16_t adc_integral  = {0};
    uint16_t adc_peak      = {0};
    uint32_t detid         = {0};
    uint32_t type          = {0};
    uint32_t algorithm     = {0};
    uint16_t version       = {0};

    std::vector<TriggerPrimitive> tp_list;
  };
```

```
namespace DuneTriggerAlgs {
  struct TriggerCandidate {
    int64_t  time_start    = {0};
    int64_t  time_end      = {0};
    int64_t  time_decided  = {0};
    uint32_t detid         = {0};
    uint32_t type          = {0};
    uint32_t algorithm     = {0};
    uint16_t version       = {0};

    std::vector<TriggerActivity> ta_list;
  };
```

Intermission (I) - Background info

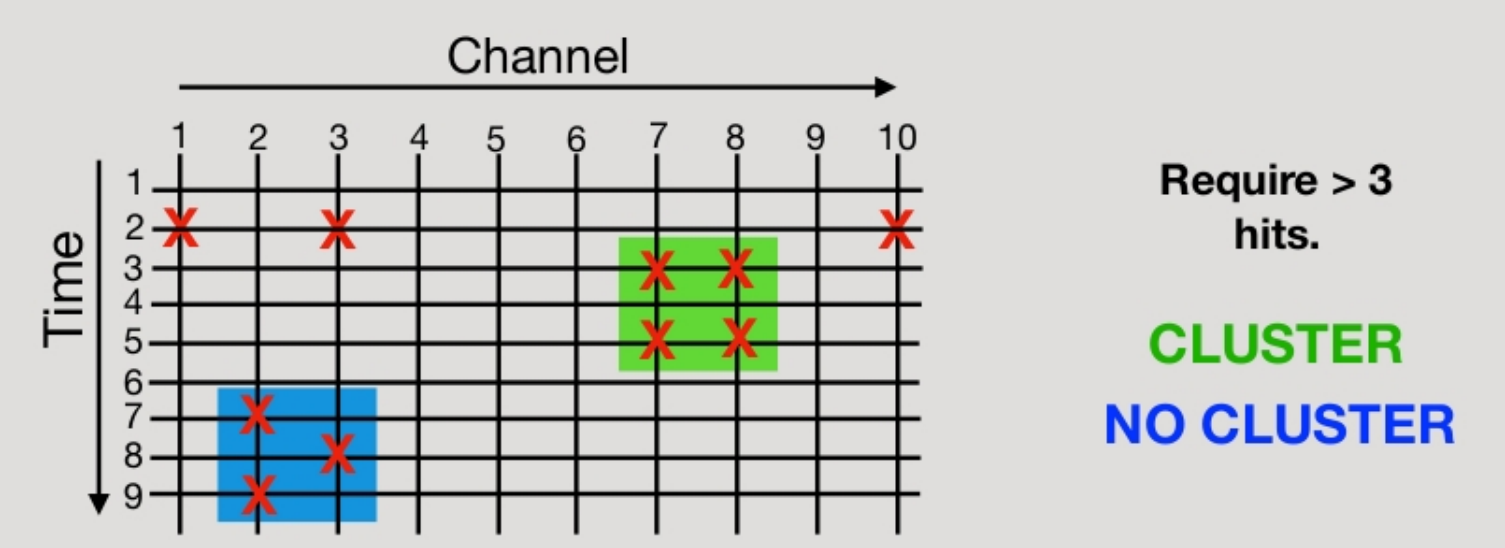
Module is a skeleton/back bone, built with a SNB approach in mind.



Brett's diagram

Intermission (II) - Basics on SNB triggering

TAs are formed by clustering TPs in time and space (channel).



At least two parameters needed:

- Channel tolerance between TPs
- Time tolerance between TPs

Additionally we also have number of TPs in a TA as a selection criteria.

Intermission (III) - Basics on SNB triggering

TCs are formed by simply counting TAs in a time window.
At least two parameters needed:

- Size of the time window
- Number of TAs in the time window

→ Example of a simple TA and TC finder:

DuneTriggerAlgs/dune-trigger-algs/Supernova/TriggerActivityMaker_Supernova.*
DuneTriggerAlgs/dune-trigger-algs/Supernova/TriggerCandidateMaker_Supernova.*

The finders has a pure virtual function, operator(), where the algorithmic part happens

Appfwk interface (DAQDuneTriggers)

- Where the DAQProcess are implemented and calling these operator() functions
- Example class (DAQDuneTriggers/plugins/DAQTriggerActivityMaker.cpp):

```
class DAQTriggerActivityMaker: public dunedaq::appfwk::DAQModule, DuneTriggerAlgs::TriggerActivityMakerSupernova {
```

- Fills and consumes queues of trigger objects (listrev based)
- Holds and parses all configurations of the algorithms previously defined.
  - For example, in case of Supernova TC maker, we might change the threshold in number of TAs at which the trigger is emitted
  - Working on going on this front for new appfwk version

Example: TC (I)

operator() call at DAQDuneTriggers/plugins/DAQTriggerCandidateMaker.cpp

```
void DAQTriggerCandidateMaker::do_work(std::atomic<bool>& running_flag) {
  TLOG(TLVL_ENTER_EXIT_METHODS) << get_name() << ": Entering do_work() method";
  int receivedCount = 0;
  int sentCount = 0;
  TriggerActivity activ;

  while (running_flag.load()) {
    TLOG(TLVL_CANDIDATE) << get_name() << ": Going to receive data from input queue";
    try {
      inputQueue_->pop(activ, queueTimeout_);
    } catch (const dunedaq::appfwk::QueueTimeoutExpired& excpt) {
      // it is perfectly reasonable that there might be no data in the queue
      // some fraction of the times that we check, so we just continue on and try again
      continue;
    }
    ++receivedCount;

    std::vector<TriggerCandidate> tcs;
    this->operator()(activ,tcs);
```

Example: TC (II)

operator() definition at

DuneTriggerAlgs/dune-trigger-algs/Supernova/TriggerCandidateMaker_Supernova.hh

```
public:
  /// The function that gets call when there is a new activity
  void operator()(const TriggerActivity&, std::vector<TriggerCandidate>&);

protected:
  std::vector<TriggerActivity> m_activity;
  std::atomic<int64_t > m_time_window   = {500'000'000};   /// Slinding time window to
  std::atomic<uint16_t> m_threshold     = {6};             /// Minimum number of activi
  std::atomic<uint16_t> m_hit_threshold = {3};             /// Minimum number of primit
```

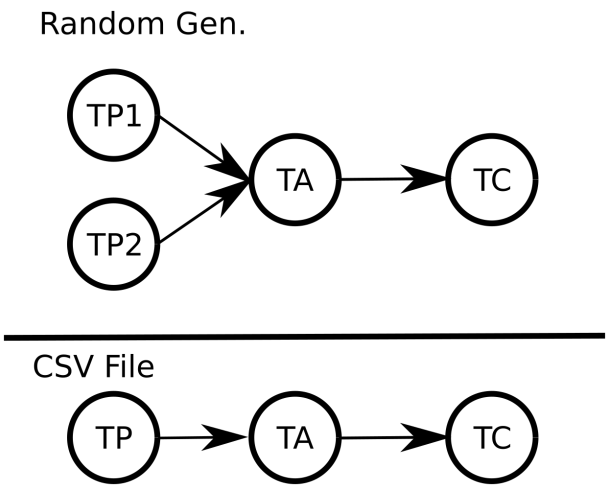
Example: TC (III)

operator() description at

DuneTriggerAlgs/dune-trigger-algs/Supernova/TriggerCandidateMaker_Supernova.cc

```
void TriggerCandidateMakerSupernova::operator()(const TriggerActivity& activity,
                                                std::vector<TriggerCandidate>& cand) {
  int64_t time = activity.time_start;
  FlushOldActivity(time); // get rid of old activities in the buffer
  if (activity.tp_list.size()>m_hit_threshold)
    m_activity.push_back(activity);

  // Yay! we have a trigger!
  if (m_activity.size() > m_threshold) {
    std::chrono::time_point<std::chrono::steady_clock> now;

    // set all bits to one (probably this will mean down the line: "record every part of the detector")
    uint32_t detid = 0xFFFFFFFF;

    TriggerCandidate trigger {time - 500'000'000, // time_start (10 seconds before the start of the activity)
                              activity.time_end, // time_end, but that should probably be _at least_ this number
                              int64_t(pd_clock(now.time_since_epoch()).count()), // this is now in dune time, with a c
                              detid, // all the detector
                              0, //type ( flag that says what type of trigger might be (e.g. SN/Muon/Beam) )
                              0, //algorithm ( flag that says which algorithm created the trigger (e.g. SN/HE/Solar) )
                              0, //version of the above
                              m_activity}; // TAs used to form this trigger candidate
    m_activity.clear();
    // Give the trigger word back
    cand.push_back(trigger);
    return;
  }
```

What to do if you want to use this module?

- You become a collaborator on the module development
  - We want to agree on a minimum version so it can be incorporated on DUNE-DAQ repository, and update conventions.

- You need to write your version for all chain (TP→TA→TC):
  - TP: Create your own random generator or csv file
  - TA: Create your maker/finder, if current not suitable (probably isn't)
  - TC: Create your triggering condition