Framework NOP C++ 4.0 (libnop)

This framwork was developed to make it easier to build applications using the NOP in C++. It uses CMake to build the library and applications.

Requirements

- Unix or Windows (OSX is untested)
- CMake 3.14
- C++20 compliant compiler (MSVC 14.2 / GCC 10 or GCC 11 is recomended)

Usage instructions

Check the template project for a ready to use template: https://nop.dainf.ct.utfpr.edu.br/nop-applications/framework-application/framework-cpp-4-application/template-application

Check libnop_gtest/test.cpp file for more examples on how to use the framework.

Build options

The following options must be set during cmake generation.

Usage options:

- LIBNOP_SCHEDULER_ENABLE: Enable use of schedluer for executing Rules (reduces performance)
- LIBNOP_LOG_ENABLE: Enable using logs (heavy performance hit)

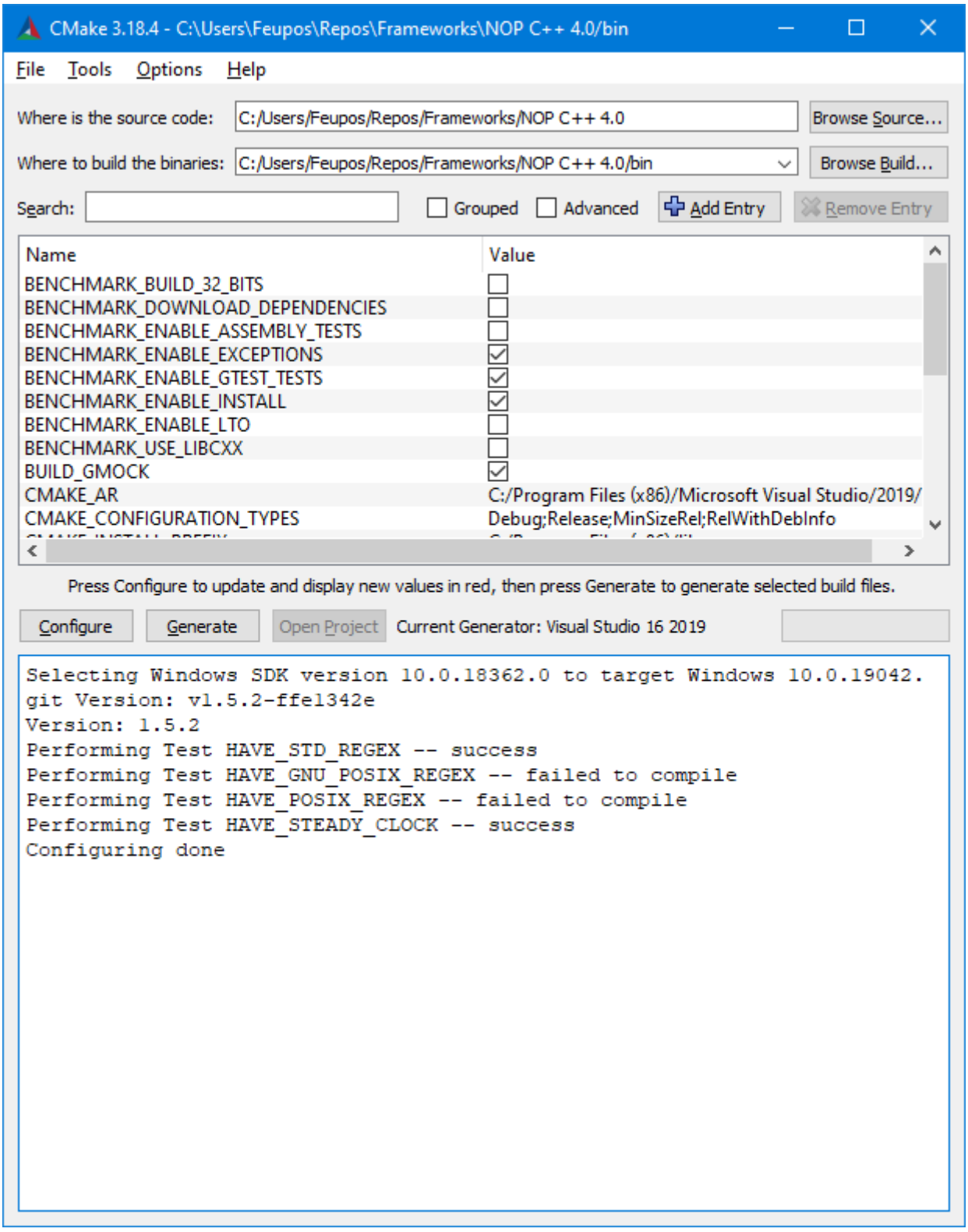
Development options:

- LIBNOP_TEST_ENABLE: Enable building unity tests
- LIBNOP_COVERAGE_ENABLE: Enable code coverage
- LIBNOP_BENCHMARK_ENABLE: Enable build benchmarks
- LIBNOP_BENCHMARK_SENSOR_ENABLE: Enable build sensor benchmark
- LIBNOP_BENCHMARK_BITONIC_ENABLE: Enable build bitonic benchmark
- LIBNOP_BENCHMARK_RANDOMFOREST_ENABLE: Enable build random forest benchmark
- LIBNOP_FW2_TEST_ENABLE: Enable build NOP C++ Framework 2.0 tests

For maximum performance, all options are disabled by default. As an alternative you can add "#define LIBNOP_SCHEDULER_ENABLE" before including "#include "NOPFramework.h"" or "#include "libnop/framework.h"" to enable the scheduler.

Hot to build (Windows)

Download and install the CMake GUI for windows and generate the project for your IDE. https://cmake.org/download/



1. Set source folder - Ex: nop-framework-cpp-4/
2. Set build folder - Ex: nop-framework-cpp-4/build
3. Configure the desired options
4. Generate
5. Open Project

Hot to build (Unix)

On Ubuntu 20.04 you must install gcc 10/11 and libtbb and cmake as follows:

1. `sudo apt-get install -y g++-11`

2. `sudo apt-get install -y libtbb-dev`
3. `sudo apt-get install python3-pip`
4. `sudo pip install cmake`

With the dependencies installed you can compile:

1. `mkdir build && cd build`
2. `cmake .. -DCMAKE_CXX_COMPILER=g++-11`
3. `sudo make install`

Tests and benchmark (Unix)

After building you can run the tests and benchmarks:

1. `libnop_gtest/libnop_gtest`
2. `libnop_gbench/libnop_gbench`

How to use

Include the header "libnop/framework.h" in your files.

To utilize this library in other `CMake` projects you can add the following lines to your CMakeLists.txt:

```
find_package(libnop)
if(NOT libnop_FOUND)
    include(FetchContent)
    FetchContent_Declare(libnop
        GIT_REPOSITORY https://nop.dainf.ct.utfpr.edu.br/nop-
implementations/frameworks/nop-framework-cpp-4.git
        GIT_TAG master
    )

    FetchContent_GetProperties(libnop)
    if(NOT libnop_POPULATED)
        FetchContent_MakeAvailable(libnop)
    endif()
endif()
```

C++20 is required for use:

```
set_target_properties(<target> PROPERTIES
    CXX_STANDARD 20
)
```

Alternatively you can clone the repository and install the library in your system:

1. `git clone https://nop.dainf.ct.utfpr.edu.br/nop-implementations/frameworks/nop-framework-cpp-4.git`
2. `cd nop-framework-cpp-4.git`
3. `mkdir build`
4. `cd build`
5. `cmke .. -DCMAKE_CXX_COMPILER=g++-11`
6. `sudo make install`

Then import with CMake with find_package:

```
find_package(libnop)
```

CI/CD

To make use of code coverage make sure you have a valid runner: https://nop.dainf.ct.utfpr.edu.br/help/ci/runners/README https://docs.gitlab.com/runner/install/

How to use

To use this framework you can just define and initialize the NOP entities as desired.

For more details about the NOP entities you may check the NOP bibligraphy, such as the paper regarding this framework available here.

All the NOP entities are defined as shared pointers, so they can be easily shared in the code.

You must declare the entity and the initialize the pointer using one of the available builders.

The following header must be included: `#include "libnop/framework;h"`.

Attribute

The Attribute has a template parameter that can be any data type. It has a single builder that receives the initial value of the attribute.

```
template <typename T>
using SharedAttribute = std::shared_ptr<Attribute<T>>;
```

Example:

```
NOP::SharedAttribute<int> at1 = NOP::BuildAttribute(1);
```

Premise

The Premise can be built with two Attributes or an Attribute and a value. The third parameter is the Premise operation between it Attributes.

```
template <typename T, typename Func>
auto BuildPremise(std::shared_ptr<Attribute<T>> lhs, T rhs, Func op);

template <typename T, typename Func>
auto BuildPremise(std::shared_ptr<Attribute<T>> lhs,
                  std::shared_ptr<Attribute<T>> rhs, Func op);

/* The operation op can be:
    NOP::Equal(),
    NOP::Different(),
    NOP::Greater(),
    NOP::GreaterEqual(),
    NOP::Less(),
    NOP::LessEqual()
*/
```

Example:

```
NOP::SharedAttribute<int> at1 = NOP::BuildAttribute<int>(-1);
NOP::SharedAttribute<int> at2 = NOP::BuildAttribute<int>(-2);

NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
NOP::SharedPremise pr2 = NOP::BuildPremise(at1, 0, NOP::Different());
```

Condition

The Condition is built using as arguments one or more Premises, Conditions (as Sub-Bonditions) or Rules (as Master Rules).

The simple condition builder with the condition type as a template parameter:

```
template <EConditionType type, typename... Args>
auto BuildCondition(Args... args);

/* The condition type can be:
    NOP::Single,
    NOP::Conjunction,
    NOP::Disjunction
*/
```

More complex conditions can be composed as a boolean expression, and passing the pointers as arguments. The operation can be passed as a lambda expression, where the CONDITION macro can be used to facilitate the definition.

```
template <typename... Args>
auto BuildCondition(std::function<bool(void)> operation, Args... args);

#define CONDITION(expression) [=, this]() { return bool(expression); }
```

Example:

```
NOP::SharedCondition cn1 = NOP::BuildCondition<NOP::Conjunction>(pr1,
pr2);
NOP::SharedCondition cn2 =
        NOP::BuildCondition(CONDITION((*pr1 || *pr2) && !*pr3), pr1, pr2,
pr3);
```

Method

The Method is stored as a std::function with no parameters. It can be initialized with a lambda expression. The macro METHOD can be used to facilitate the definition.

```
using Method = std::function<void(void)>;
#define METHOD(expression) [&]() { expression }
```

Example:

```
int counter{0};
NOP::Method mt1{ METHOD(counter++;) };
```

Instigation

The Instigation is built with one or more Methods as parameters. A template parameter can be used to define a parallel Instigation (all Methods are executed in parallel).

```
template <NOPFlag Flag = Default, typename... Args>
auto BuildInstigation(Args... methods)
```

Example:

```
NOP::SharedInstigation in1 = NOP::BuildInstigation(mt1, mt2);
NOP::SharedInstigation in2 = NOP::BuildInstigation<NOP::Parallel>(mt3,
mt4);
```

Action

The Action is built with one or more Instigations as parameters. A template parameter can be used to define a parallel Action (all Instigations are instigated in parallel).

```
template <NOPFlag Flag = Default, typename... Args>
auto BuildAction(Args... instigation);
```

Example:

```
NOP::SharedAction ac1 = NOP::BuildAction(in1, in2);
NOP::SharedAction ac2 = NOP::BuildAction<NOP::Parallel>(in3, in4);
```

Rule

The Rule is build using an Action and a Condition. When using the Scheduler you can also define a priority value (bigger values has higher priority).

```
std::shared_ptr<Rule> BuildRule(std::shared_ptr<Condition> condition,
                                std::shared_ptr<Action> action,
                                const int priority = 0)
```

Example:

```
NOP::SharedRule rl1 = NOP::BuildRule(cn1, ac1);
NOP::SharedRule rl2 = NOP::BuildRule(cn1, ac1, 10);
```

FBE

The FBE class can be inherited by any classes containing Attributes or Methods. It is not required, but facilitates organizing the NOP entities.

```
class Example: public NOP::FBE
{
    NOP::SharedAttribute<int> at1;
};
```

Scheduler

The Scheduler can be used to ensure determinism when executing the Rules. It executes Rules one by one, while rechecking the condition for each one. Beware, performance is heavily

impacted when using the Scheduler.

To use the Scheduler you must use the compile definition -DLIBNOP_SCHEDULER_ENABLE-=ON or define `#define LIBNOP_SCHEDULER_ENABLE` before including `#include "libnop/framework;h"`.

The Scheduler execution strategy can be set at any point. The ignore_conflict flag may be used if conflict resolution is not desired.

```
void SetStrategy(EStrategy strategy, bool ignore_conflict = false);

/* Valid strategy types are:
NOP::None,
NOP::FIFO,
NOP::LIFO,
NOP::Priority
*/
```

Example:

```
NOP::Scheduler::Instance().SetStrategy(NOP::FIFO);
```

Logger

The framework provides debuggin facilities by means of a Logger.

The Logger can be used to log the NOP execution flow. Beware, performance is heavily impacted when using the Logger.

To use the Scheduler you must use the compile definition -DLIBNOP_LOGGER_ENABLE-=ON or define `#define LIBNOP_LOGGER_ENABLE` before including `#include "libnop/framework;h"`.

The log file location can be set with:

```
void Logger::SetLogFileName(const std::string& file_name);
```

All the NOP entities proved auxiliary builder that can name the entities for logging purposes. Those builders are prefixed Named, and have an additional name parameter. The FBE constructor also accepts a name parameter. When those entities are members of an FBE they may also pass the FBE pointer as a parameter.

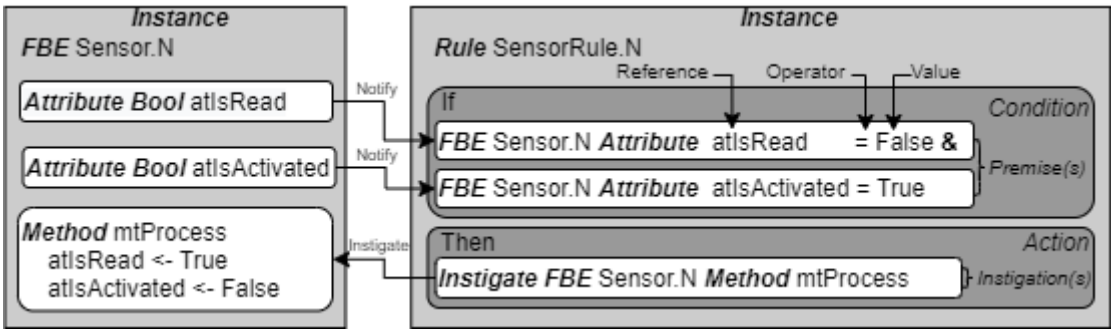
Example:

```
NOP::Logger::Instance().SetLogFileName("test.log");
```

```
struct Test : NOP::FBE
{
    explicit Test(const std::string_view name) : FBE(name) {}
    NOP::SharedAttribute<int> at1 =
        NOP::BuildAttributeNamed("at1", this, -1);
    NOP::SharedAttribute<int> at2 =
        NOP::BuildAttributeNamed("at2", this, -2);
    NOP::SharedPremise pr1 =
        NOP::BuildPremiseNamed("pr1", this, at1, at2, NOP::Equal());
    NOP::SharedCondition cn1 =
        NOP::BuildConditionNamed<NOP::Single:("cn1", this, pr1);

    std::atomic<int> executionCounter{0};
    NOP::Method mt = [&] { executionCounter++; };
    NOP::SharedInstigation in1 =
        NOP::BuildInstigationNamed("in1", this, mt);
    NOP::SharedAction ac1 = NOP::BuildActionNamed("ac1", this, in1);

    NOP::SharedRule rl1 = NOP::BuildRuleNamed("rl1", this, cn1, ac1);
};

Test test{"TestFBE"};
```

FBE example

The following FBE and Rule can be implemented as an example.



As described using NOPL:

```
fbe Sensor
  public boolean atIsRead = false
  public boolean atIsActivated = false
  private method mtProcess
    attribution
      this.atIsRead = true
      this.atIsActivated = false
    end_attribution
  end_method
  rule rlSensor
    condition
        premise prIsActivated
          this.atIsActivated == true
```

```
        end_premise
        and
        premise prIsNotRead
          this.atIsRead == false
        end_premise
    end_condition
    action sequential
        instigation sequential
          call this.mtProcess()
        end_instigation
    end_action
  end_rule
end_fbe
```

And defined using the framework:

```
struct NOPSensor: NOP::FBE
{
    NOP::SharedAttribute<bool> atIsRead{NOP::BuildAttribute(false)};
    NOP::SharedAttribute<bool> atIsActivated{NOP::BuildAttribute(false)};
    NOP::Method mtProcess{METHOD(atIsRead->SetValue(true); atIsActivated-
>SetValue(false);)};

    NOP::SharedPremise prIsActivated{
        NOP::BuildPremise(atIsActivated, true, NOP::Equal())};
    NOP::SharedPremise prIsNotRead{
        NOP::BuildPremise(atIsRead, false, NOP::Equal())};

    NOP::SharedRule rlSensor{NOP::BuildRule(
        NOP::BuildCondition<NOP::Conjunction>(
            NOP::BuildPremise(atIsActivated, true, NOP::Equal()),
            NOP::BuildPremise(atIsRead, false, NOP::Equal())
            ),
        NOP::BuildAction(
            NOP::BuildInstigation(mtProcess)
            )
        )
    };
};
```