THE UMA-SOM PLATFORM FOR GENERAL LEARNING

– A USER'S MANUAL –

DAN P. GURALNIK† AND SIQI HUANG

ABSTRACT. This technical report supplements a document by the same name describing the research and development effort by our group on a partially distributed, GPU-accelerated implementation of a modality-agnostic architecture for life-long general learning, based on the Universal Memory Architecture (UMA) memory model previously introduced by Guralnik and Koditschek. The design by Huang and Guralnik emphasizes modularity, distributability and flexibility of application, by separating the computational components of the architecture (implemented in CUDA) from the RESTful communication framework (C++) and the simulation state update and environment (Python).

This document is envisioned both as a user's manual and as a technical description of the architecture in support of further development efforts.

1. Preparing, Inspecting and Applying the UMA-SOM Package

This section focuses on preparing the UMA-SOM system for work without paying much attention to system design. A minimal overview of the design at the highest level is provided to help the user follow the installation process and understand proper use. Knowledge of the parent document [1]—specifically Section 3.1 therein—is assumed for the purpose of reading Section 1.5, discussing the writing of simulation scripts.

1.1. **Building the UMA Package.**

The UMA project consists of the following parts:

(1) UMA Simulation Engine (Python),

(2) UMA Client (Python),

(3) UMA Core Service (C++).

The UMA Core Service is written in C++ and needs to be built by Visual Studio (on Windows) or by `gcc` (on Linux). UMA Core Service supports either a Windows or Linux build, and has been tested on Win7, Win10 and Ubuntu 16.04 LTS. UMA Core Service has three additional library dependencies, they are: `gtest` (Google Test), `cuda` (currently NVIDIA CUDA 8.0) and `cpprestsdk` (Microsoft Casablanca RESTful Interface). The `cuda` and `cpprestsdk` libraries require installation prior to building the project (e.g., using `apt-get` on Linux, or `vcpkg` on Windows); the `gtest` library is built in tandem with the UMA project, using a provided batch/shell script, depending on the OS. We also make use of the `Cmake` utility to streamline the build process for the user.

Detailed instructions for building are given in the project repository `readme.txt` file.

† University of Pennsylvania, Electrical & Systems Engineering Department, Philadephia, PA.
Date: March 31, 2019.

1.2. **Directory Trees.**

1.2.1. *Directory Tree: `UMA Core`.*

The UMA Core project is organized into the following folders and files:

- `[root]/`

  Here you will find batch/bash scripts named `set_env` for setting up environment variables. These should be run before each simulation to ensure proper communication between the system's components.

- `bin/`

  Contains the executables of the UMA Core project. In addition to the two sub-folders `ini/` and `log/`, a successful build will result in executable and shared library files being written to the `bin/` folder.

  - `ini/`

    The `ini/` folder under `bin/` contains all the parameter specifications necessary for launching the UMA program. Please refer to 2.2.1 for more details.

  - `log/`

    This folder will appear once UMA is run for the first time. Separate execution logs for each component of the project will be written to this folder at run time.

  - `UMA.exe`

    This is the UMA core executable files, which is generated after a successful build.

  - `UMAc.exe`

    The executable `UMAc.exe` is an additional wrapper to run `UMA.exe`, providing better CLI support. Please refer to Section 1.4.2 for details.

  - `UMAUtilTest.exe, UMAKernelTest.exe, UMACoreTest.exe, UMARestTest.exe`

    These are unit tests for the various UMA Core modules. They are built together with the main project. It is recommended to run these tests before launching an UMA simulation.

  - `*.dll(.so)`

    There are several shared libraries in the bin folder, used by the UMA Core at run time.

- `dependency/`

  The source code of this library is built by the shell script `build_dependency.bat(sh)`. The purpose of this folder is to house all third-party libraries used in the project:

  - Google Test (`gtest`), used by the C++ unit testing component.

  All the executables generated under `bin/test/` depend on this library.

- `src/`

  Contains the C++ source code. Each sub-folder houses a C++ module, except for the `common/` directory:

- `common/`
  Frequently used headers reside here.

- `utility/`
  Contains all utility functionalities in the project. The corresponding C++ module is `UMAUtil`.

- `kernel/`
  Contains CUDA kernel code and other functionalities requiring a GPU. `UMAKernel` is the resulting C++ module.

- `RESTAPI/`
  Contains the entry of the project (main function) and the files that construct the REST API callable from the Python client. Its C++ module is `UMAc`.

- `RESTful/`
  Includes the basic REST functionality used in the project, wrapped around `cpprestsdk`. Its C++ module is `UMARest`.

- `core/`
  Source code for the module `UMACore`, implementing BUAs, their associated data structures and functionalities.

- `launcher/`
  Source code for the module `UMA`, providing a CLI interface to interact with UMAc. The CLI interface includes `start`, `stop` and `status`.

- `kernelTest/`
  Unit test files for module `UMAKernel`. Its C++ module is `UMAKernelTest`.

- `utilityTest/`
  Unit test files for module `UMAUtil`. Its C++ module is `UMAUtilTest`.

- `coreTest/`
  Unit test files for module `UMACore`. Its C++ module is `UMACore`.

- `RESTfulTest/`
  Unit test files for module `UMARestTest`. Its C++ module is `RESTfulTest`.

- `CMakeLists.txt`
  Configuration file for the CMake installer of the package.

- `deployment/`
  This directory contains scripts to manage a deployment of multiple UMA cores.

  - `cluster/`
    The `UMA Core` instances are deployed into this directory.

- `cluster_setting.py`
  Contains common settings for deploying multiple `UMA instances`.

- `deploy.yml`
  The `yaml` file stating the number of UMA instances to be deployed, as well as the host and port addresses of these instances.

- `deploy_UMA.py`
  A python script to be invoked for proper deployment of multiple `UMA Core` instances.

- `run_UMA.py`
  Python script for launching all deployed `UMA Core` instances (under `cluster/`). After calling it, the `pid` of each `UMA Core` instances will be recorded for use by `stop_UMA.py`.

- `stop_UMA.py`
  Python script for stopping all deployed running `UMA Core` instances. It terminates all the `UMA Core` instances whose `pid`s were recorded by `run_UMA.py`.

- `build_dependency.bat`
  Under Windows, this must be run to build the code contained in `dependency/`.

- `build_dependency.sh`
  Under Ubuntu, this needs to be run to build the code under `dependency/`.

- `build_vs.bat`
  This batch file is used on Windows only, to build the Visual Studio solution `UMA.sln`.

- `build_kernel.sh`
  This shell script is used on Linux only, to build *only* the `UMAKernel` module (on Windows, `UMAKernel` is built by Visual Studio together with the other modules).

- `readme.txt`
  Detailed instructions for preparing and running the build.

1.2.2. *Directory Tree: `UMA Client`.*

- `[root]/`
  Here you will find batch/bash scripts named `set_env` for setting up environment variables. These should be run before each simulation to ensure proper communication between the system's components.

- `bin/`
  Contains standard python executables running the UMA client, such as a simulation launcher.

- `lib/`
  The folder contains several libraries that are used in UMA client, including a `client` module for direct REST communication with UMA Core, a `cluster` module for launching multiple client instance simultaneously, `perf` module for gathering performance metrics, and `qa` module for functional test

utility.

- `pylib/`
  Contains the Python libraries for the UMA front end:

  - `UMA/`
    Libraries defining the UMA architecture data structures—Signals, Experiments, Agents—and some supporting functions. These are directly included by simulation scripts in the `suites/` folder. Their scope does not extend beyond the Python side.

  - `Client/`
    In contrast, the `Client/` sub-folder contains Python scripts running outside the scope of Python. These are responsible for connecting to the core services.
  - `perf/` Finally, the `perf/` sub-folder contains libraries for performance testing.

- `suite/`
  Contains the available Python simulation scripts using the `UMACore` service:

  - `sniffy/`
    See [1], Section 4.1.1 for a detailed description of this environment.

  - `mice/`
    See [1], Section 4.1.2 for a detailed description of this environment.

  - `test/`
    Contains all functional and performance test scripts.

1.3. **Deployment.** An UMA deployment includes two parts: the UMA Python front end and the UMA Core (C++/CUDA back end), see Figure 1.

UMA Python modules can run on any platform supporting Python 2.7 (preferably an up-to-date Anaconda distribution); the UMA Core, for now, only supports Windows and Linux. Refer to Section 1.1 for details on building the UMA core.

For small simulations and/or debugging purposes (for example, see [1], Section 4.2.1), we recommend to run UMA Core on Windows, and build with MSVC. UMA Python modules should be run on the same machine, and use `localhost` as their server.

For larger scale deployment, it is preferable to build and deploy the UMA core on Linux, and launch UMA Python on a separate instance (Linux also preferred), residing in the same region, to maintain low network traffic latency. AWS would be a desirabe place to host the environment.

The UMA Core application is currently only capable of running as a single instance: running multiple UMA instances and having them communicate with each other is not yet supported. However, this distributed functionality is easily achievable, due to the UMA REST functionality, by altering the UMARest module, for it to both handle and send requests to other UMA instances.

1.3.1. *Parallel execution of multiple simulation runs.* In addition, it is possible to run several UMA Core instances under the same host with different ports, for example, to execute several distinct runs of the same experiment in parallel. For this purpose, we look into the environment setup under the `deployment/` folder:

- The number of `UMA Core` instances must be specified in the `deploy.yml` file (see details below).

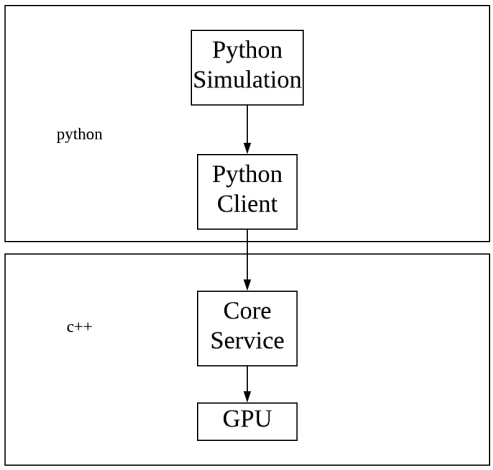

FIGURE 1. Separation of the UMA packages into a Python front end and a C++/CUDA back end.

- Run `deploy_UMA.py` to copy several `UMA Core` instances into the `cluster` folder.
- It is now possible to launch all the deployed `UMA Core` instances by executing `run_UMA.py`, or stop them by executing `stop_UMA.py`.
- To change deployment parameters (or to reset the deployment), simply alter `deploy.yml` as needed and re-run `deploy_UMA.py`.

The number of UMA instances to launch is controlled by `Ninstances` in `deploy.yml`. The deployed UMA instances will use consecutive ports starting from `port` defined in `deploy.yml`.

When launching `UMA Client` instances with `UMA Test Launcher`, the `Ninstances` value provided in `lib/cluster/cluster.yml` defines the UMA instances to connect to, and `port` defines the starting point for those UMA instances. If the launcher is supposed to use the current deployment, these values need to be compatible. In particular, please make sure the number of UMA instances actually deployed (and running) is no less than the number of instances requested in `UMA Client`. For more information please refer to Section 1.4.2.

1.4. **UMA Client.** UMA Client is the python package used to interact with UMA Core, launching uma tests, and testing uma functionality. It uses the `requests` python package to communicate with UMA Core through a REST API. We maintain it in a separate repository, separate from UMA Core. A review of UMA Client components follows.

1.4.1. *UMA REST client.* The UMA REST client is located in `lib/client/UMARest.py`, and it consists of several object wrappers, which map to the `UMA Core` objects in the c++ program. Every wrapper class has a `service` object, responsible for maintaining the connection to `UMA Core`. In addition, each object contains all the ids of its ancestors. We use this strategy to pinpoint the resource in `UMA Core`. Please refer to Section 2.7.2 for details.

1.4.2. *UMA Test Launcher.* To run an UMA simulation, it is possible to call the REST API directly in your python script. However, an easier way is to fully utilize the power of the module `UMA REST Client` module, and invoke it with `UMA Test Launcher`. Although UMA Test Launcher and `UMA Rest Client` are separate modules, we still recommend using them in tandem, because together they provide means for achieving parallel execution of multiple runs of an UMA simulation. The `UMA Test Launcher` has files both under `bin/` and `lib/`.

The module `bin/launcher.py` launches an UMA simulation using an input `yaml` file. The launcher requires the `yaml` file to contain the following fields:

- `script`:
  The Python simulation script to run, in the scope of `UMA Simulation Engine`.

- `func`:
  The function in the `script` describing a single run.

- `first_run`:
  Number to assign to the first run in the simulation. Normally, this would be set to zero, except if a previously stopped simulation needs to be continued.

- `Nruns`:
  The number of runs required for the current simulation. Each run will generate a different `experiment_id` to prevent confusion in the REST addressing of the different UMA Cores.

- `para`:
  The parameters you want to pass in for the `func` to run. It will be received as a dictionary, and the function that you invoke can get those parameters and utilize them.

The `UMA Test Launcher` has another component under `lib/cluster`, consisting of two `yaml` files: `pool.yml` is responsible for telling `UMA Test Launcher` how many client processes are allowed to run in parallel; `cluster.yml` sets the number of `UMA Core` instances that `UMA Test Launcher` may connect to, and the corresponding host and port addresses. There are also two `python` files: `cluster.py` is the library used for scheduling the distinct simulation runs, and `cluster_setting.py` stores some common variables and functions used in `UMA Test Launcher`.

1.4.3. *UMA Test Suite.* A number of unit tests and functional tests are put under `suites/test`. To run the unit test, just run `suites/test/unittest/run_test.py`. It will use the data provided under `suites/test/unittest/data` and the library under `lib/client/test_wrapper.py`. To run the functional tests, just go to `suites/test/functional/tests/`, and choose the test you would like to launch: enter its folder and use `uma_test.py` to run the test. The `uma_test.py` resides under `bin/` and is only used for launching functional tests.

1.5. **UMA simulation Scripts.**

1.5.1. *Running a simulation script: general principles.* The current system is very flexible: any number of UMA cores may be run to process any number of parallel simulation threads (depending on system resources, of course). To clarify the operation of the system, we outline general principles here, but **we strongly recommend using the launcher** (Section 1.4.2), even for single-run simulations, despite the overhead of preparing the required `Yaml` file.

- Running an UMA simulation currently requires a Python script to import the UMA-SOM engine `som2_noEP.py` from the `pylib/UMA/` project directory, as well as the client modules from `pylib/Client/`. We recommend adding these to the $PYTHONPATH environment variable. Alternatively, these modules may be copied into the working directory.
- The UMA core `UMAc(.exe)` executable needs to be run prior to executing the simulation script. The directory containing it must also contain any other items present in the `bin/` directory of the UMA Core project at the end of the build process. Exceptions to this requirement are the `bin/test/` directory (you may not be interested in testing) and the `bin/log/` directory (logs are created at run time).
- Please make sure the log files (of the UMA core) are purged from time to time (see also initialization files settings, Section 2.2.4).

1.5.2. *Designing a new simulation script.* An UMA simulation operates within the framework of an Experiment Python object, whose role is to maintain the simulation state and to communicate with the BUAs through `Agent` objects. Section 3.1 of the technical report provides a detailed discussion of the update procedure, implemented by the `update_state` method of the `Experiment` class. Technical details from a more advanced developer's perspective are discussed in Section 2.7. The bare essentials to be aware of are:

Experiment: An `Experiment` object must be constructed, and it is convenient to create a reference to the decision stack, as well as to the experiment clock, which are reserved ids in the Experiment:

- `EX=Experiment()`
- `id_dec='decision'`
- `id_count='counter'`

Registration: Each state variable—or *measurable*—must be registered before it could be addressed by any other component of the experiment:

- <*id_meas*>`=EX.register(`<*meas_name*>`)`

When registering a measurable, the user provides a unique ID tag (MID for short), to be used for referring to values of this measurable in the experiment state. If a tag is not provided, a *uuid* is issued an used by the system instead. Binary sensors—those corresponding to the control values provided by BUAs included—are issued MIDs in pairs, using:

- <*id_meas*>`,`<*cid_meas*>`=EX.register_sensor(`<*meas_name*>`)`

Sensors *must* be registered in this way (rather than as two separate MIDs).

Experiment State: The experiment state is maintained as a dictionary of deques[1], `EX._STATE`, indexed by the registered MIDs. The experiment state may be accessed as follows:

- `EX.state_all()`: provides access to the entire state dictionary;
- `EX.this_state(`<*mid*>`,`<*delta*>`=0)`: returns the state of measurable <*mid*>, <*delta*> cycles ago (default is the current state);
- `EX.set_state(`<*mid*>`,`<*val*>`)`: push <*val*> to the head of the deque for measurable <*mid*> (sets the current state).

Supplying Definitions: Each measurable requires a definition of its state update function as a function of the state dictionary. An appropriate function object must be constructed, and passed as an argument to the relevant constructor, as detailed below.

Constructing Measurables: Plain measurables are constructed using:

- `EX.construct_measurable(`<*mid*>`,`<*definition*>`=None,`<*init_value*>`=None,`<*depth*>`=1,`<*decdep*>`=False)`: A measurable with MID <*mid*> is added to the experiment:

(1) The provided MID *must* be registered before this function is called.
(2) <*definition*> is a function object taking a single input, named `state`, envisioned as a function of the experiment state dictionary.
(3) <*init_value*> is a list with which to initialize this measurable's entry in the experiment state dictionary.
(4) <*depth*>+1 is the desired bound on the length of the deque representing <*mid*> in the experiment state dictionary (default is 2, to store the current state and the preceding state).
(5) <*decdep*> is a Boolean value indicating whether <*mid*> will participate in the decision sweep (see [1], Section 3.1.2).

Constructing Sensors: Following the same syntax as the constructor for measurables, the constructor for sensors is:

- `EX.construct_sensor(`<*mid*>`,`<*definition*>`=None,`<*init_value*>`=None,`<*depth*>`=1,`<*decdep*>`=False)`:

The only difference in functionality is that this constructor sets up *two* measurables: one with MID <*mid*>, and one with MID <*mid*>`+'*'`, corresponding to the negated sensor.

[1] `https://docs.python.org/2/library/collections.html#collections.deque`

Constructing Agents: BUAs, in their dual role of both decision-makers and sensors, require a separate constructor:

- `BUA=EX.construct_agent(<`*mid*`>,<`*signal*`>,<`*definition*`>,<`*params*`>)`

The input entries for the constructor are:

(1) <*mid*> is the same MID used to register the *sensor* corresponding to the BUA being constructed;
(2) <*signal*> is a non-negative real-valued measurable (whose MID should be registered when the constructor is called), coinciding with the BUA's *motivational signal* (see [1], opening of Section 2.1.2);
(3) <*definition*> is the BUA's sensor update function;
(4) <*params*> are the BUAs initialization parameters: currently, a pair (Q, AT), where $Q \in (\frac{1}{2}, 1)$ is the discount parameter and $AT =$ `True`/`False` depending on whether the BUA will be auto-targeting or not (see [1], Section 2.1.2, stage (6) of BUA decision cycle).

Assigning Sensors to Agents: Once an Agent class object `BUA` has been constructed, it is possible to assign sensors to it:

- `BUA.add_sensor(<`*mid*`>)`

Assigns the sensor <*mid*> to both snapshot structures of Agent `BUA`. Scripted sensors need to be assigned prior to initialization (see below).

After `BUA` has been initialized, a different syntax is available:

- `BUA.add_sensor(<`*mid*`>,<`*token*`>=None)`

The sensor is assigned to the snapshot specified by <*token*> (allowed values `'plus'`, `'minus'` or unspecified, which defaults to the active snapshot). This is not intended for use in simulation scripts, and will interfere with proper function if used after `BUA` has been initialized (see below).

Initialization: Every BUA needs to be set up in the UMA core before it can be used. This is done using the instruction:

- `BUA.init()`

No scripted sensors may be added to `BUA` after this instruction is given.

State Update: Once all BUAs have been initialized, one is free to advance the simulation clock. This is done by a call of the form:

- `EX.update_state(<`*instruction*`>='decide')`

where <*instruction*> is either absent (equivalently, explicitly set to `'decide'`), or is required to be a list of MIDs containing, for each scripted BUA, either its MID or the MID of its negation (but not both). When present, the instruction will be copied over the decisions produced by the scripted agents during the decision sweep (so as not to prevent any further deliberation such as—but not limited to—hard-wired arbitration of conflicting actions).

1.5.3. *Collecting Data from an Experiment.* An `experiment_output` object for recording state information from an experiment `EX` is provided via:

- `REC=experiment_output(EX,<`*preamble*`>,unbufferedQ=True)`

Once constructed, the recorder `REC` will create two data files with the name specified in the input dictionary <*preamble*> followed by `.dat` (the "data file") and `.supp` (the "supplementary file") extensions. It will also open a preamble file by the same name, followed by `.pre`, where the preamble dictionary will be recorded for future use.

Recorder Member Functions: The member functions of a `experiment_output` object are:

- `record()`

  Creates a JSON-serialized data record and saves it to the data file. This should be used once in each update cycle (more below).

- `addendum(<`*tag*`,<`*item*`>)`

  Adds a data item of the form *tag:item* to the supplementary data file. These records are meant to contain information relevant to the simulation run as a whole.

- `close()`

  Saves and closes the data files.

The Preamble file: The `.pre` file produced by `REC` is JSON-serialized dictionary extending the preamble by adding the following items:

- `'agents'`

  An ordered list of the BUAs in the experiment.

- `'mids_recorded'`

  An ordered list of the MIDs recorded from `EX` at each update cycle (or whenever the `record()` command is issued). Each record in the `.dat` file produced by `REC` is an ordered list of values of these MIDs, given in the same order. The list is produced by copying an *existing* item in the preamble, tagged with `'mids_to_record'`; if the latter is absent, the list will default to `EX._MID`, the full list of MIDs of the experiment.

- `'ex_data_recorded'`

  An ordered list of keys corresponding to data collected by the experiment during the update cycle (`EX.update_state(...)`). This list is populated automatically, unless the preamble specifies `'ex_dataQ'` to be `False`, in which case the list remains empty and no recording takes place. Current recorded values are:

`'entering_update_cycle'`	Time stamp
`'exiting_update_cycle'`	Time stamp

- `'agent_data_recorded'`

  An ordered list of keys corresponding to internal data collected from each BUA during the update cycle. This list is populated automatically, unless the preamble specifies `'ex_agentQ'` to be `False`, in which case the list remains empty and no recording takes place. Current defaults are:

`'entering_decision_cycle'`	Time stamp
`'activity'`	reports the active snapshot ('plus'/'minus')
`'deliberateQ'`	reports whether or not the BUA's decision was deliberate or randomized
`'size'`	reports the max of the BUA's snapshot sizes
`'exiting_decision_cycle'`	Time stamp
`'override'`	reports an instruction override if there is one (empty string if none)
`'final'`	reports the final decision (post-arbitration) on the BUAs actuation bit value.

The Data File: The `.dat` file produced by `REC` is a sequence of JSON-serialized dictionaries with the following structure:

- `'mids_recorded':`

  An ordered list of values of the recorded MIDs.

- `'ex_data_recorded'`

JSON-serialized dictionary of the data recorded from the experiment's update cycle.

- `'agent_data_recorded'`
  JSON-serialized dictionary in the format `agent_id:data`, where the data is the data collected for BUA `agent_id` in `EX.update_state(...)`.

The Supplementary Data File: The `.supp` file produced by `REC` is a JSON-serialized dictionary whose contents (of the form *tag:item*) may be arbitrary.

BUA Internal State Reporting: To facilitate the gathering/use of BUA internal state information at run time, an internal state reporting function was provided in the REST Client interface:

- `ACC=UMAClientData(EX._EXPERIMENT_ID,bua_id,snapshot_id,EX._service)`

A number of requests for specific data were implemented (please consult the UMARest client code), but our experience leads us to recommend using only the simplified call for all available data:

- `ACC.get_all()`

because, despite the higher volume of data transfer, doing so minimizes the frequency of REST calls per update cycle serviced by an UMA core.

Tweaking the State Reports: While most of the internal state of any BUA could be reported, any reporting puts an additional and unnecessary load on the communication framework. **For this reason, we recommend setting the 'ex_dataQ' and 'ex_agentQ' flags to `False` whenever possible.** Constructing alternative `'ex_data_recorded'` and `'agent_data_recorded'` reports may be done by carefully altering the method `EX.update_state` of the module UMA Python module, `som2_noEP.py`. The user should feel encouraged to stick to the established format, in the form of the dictionaries (1-2) above, while avoiding significant alterations to the core code of the `EX.update_state` method.

1.5.4. *Altering the UMA Python components.* Due to the global nature of the experiment state update, at this point, *the code in* `EX.update_state` *is extremely fragile.* Any alterations to the code beyond what is necessary for collecting the required data about the experiment requires a thorough understanding—not just of the algorithms—but also of the code base, at least as discussed in the next section.

2. Design and Function of the UMA-SOM Package

2.1. **The UMA Architecture.** Recalling Figure 1, we review the top-level components of the UMA architecture:

Python Simulation Engine, or "Front End" (FE): The Python simulation engine consists of the UMA Python modules (currently `som2_noEP.py`), and of the user's executable simulation script. They former resides under `pylib/UMA/` and some test scripts reside under `suite/`. Details on how to use the simulation engine are provided in Section 1.5 above, while the function of the UMA Python modules is reviewed in detail in [1], Section 3.

UMA Python Client: The Python client connects the simulation engine to the UMA core service. The source files reside under `pylib/client/`. The files are separated according to functionality, for example: to request service from an Agent object, use `Service_Agent.py`; if a snapshot object is required to provide service, call `Service_Snapshot.py`. Further details are discussed in Section 2.7.1.

UMA Core Service, or "Back End" (BE): Written in C++, the UMA Core service supports CRUD (create/read/update/delete) of UMA objects (see Section 2.2); implementing BUA functionalities[2] (snapshot maintenance, inference and control); customized logging; and handling/throwing exceptions. The core service runs both on CPU and GPU. We cover the CPU/GPU data exchange in Section 2.3. The FE/BE communication protocol is discussed in Section 2.4.

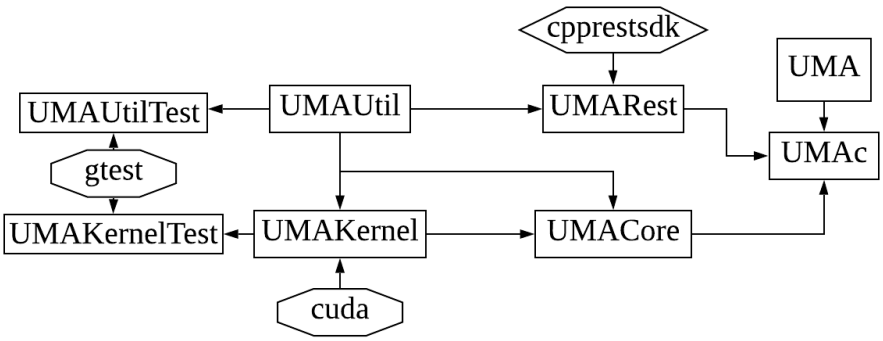

Figure 2. Package dependencies within the C++ UMA core.

2.2. **UMA C++ Core.**

2.2.1. *C++ Modules.* The UMA core services consist of seven C++ modules, whose dependency structure is summarized in Figure 2:

(1) `UMAUtil.a(.lib)`

`UMAUtil` is a static library containing basic utility functions; the logging system; exceptions; a configuration file reader; and `UMAObject` property maps and pages, all of which are consumed across the entire project. Every UMA module includes `UMAUtil` as a base library. We recommend implementing any additional common functionalities under UMAUtil, if they are to be used often. The module itself only relies on the C++ library `std`, with no other dependencies. Hence, this module must be

[2]See Section 2 of [1] for details.

built first.

The `UMAUtil` library consists of the following modules:

`ConfReader`: A helper utility function for reading configuration files.

`UMAException`: Contains all UMA exception types, and they are specific to a response code when the exception is raised by a REST call.

`Logger`: The logger class that are consumed by all other UMA modules, with adjustable log level in configuration files.

`PropertyMap`: A key-value pair map owned by every UMA object, which are used to store its own properties.

`PropertyPage`: A map of `string` to `PropertyMap`, used to store several PropertyMaps from a configuration file.

`UMAUtil`: The helper utility `namespace` that is already used widely in other UMA modules. It includes the `string` and `signal` utilities, the `system` utility and the `array` utility.

(2) `UMAUtilTest(.exe)`

`UMAUtilTest` is an executable containing the unit test cases for the `UMAUtil` module. It depends on the `UMAUtil` and `gtest` modules. New test cases should be added here whenever a new functionality is implemented in `UMAUtil`.

(3) `UMAKernel.a(.lib)`

`UMAKernel` is a static library which contains all our CUDA code (for running on a Nvidia GPU device). The module consists of pure c++ files (`.cpp`), cuda files (`.cu`), and cuda headers (`.cuh`). All are compiled under `nvcc`, and later linked against `UMACore.so(.dll)`. The `UMAKernel` library has the following modules:

`dataUtil`: The data utility module servers the host-device memory management.

`deviceUtil`: The device helper module that run on device.

`kernelUtil`: The kernel utility function, that includes many signal utility and matrix operations.

`umaBase`: Collects those functionalities that are called directly from UMACore module. All functions here are specific to UMA operation.

All CUDA related syntax throughout the project is included here, including CUDA headers. The purpose of keeping this convention is to ensure `UMAKernel` is the only CUDA-dependent module.

Note that, despite calls to `UMAUtil` functionalities in `UMAKernel`, the latter may still be compiled separately because it is a static library.

(4) `UMAKernelTest(.exe)`

`UMAKernelTest` is an executable containing unit test cases for the `UMAKernel` module. It depends on `UMAKernel`, `gtest` and `UMAUtil` (see preceding item).

(5) `UMACore.so(.dll)`

`UMACore` is a shared (dynamic linking) library, that contains all UMA simulation objects. The objects in `UMACore` are consistent with the UMA simulation entities, so that UMA Python Simulation can operate directly on those objects by using REST API. The module uses `UMAKernel` functionalities but employs no CUDA syntax. As a result, only the host compiler is required for this module, with no dependency on CUDA.

The object classes defined in this module are key to the entire project, and we review them here:

`CoreService:` the `Singleton` class responsible for managing the UMACore properties from configuration files.

`UMACoreConstant:` A `namespace` holding the definition of the UMA object type, as well as associating different `Agent` types with the appropriate `Snapshot` types.

`UMACoreObject:` The base class for all UMACore objects. It includes common variables for all UMA objects such as `uuid` and `ppm`.

`World:` A global unique `Singleton` object containing an UMA `Experiment`. It is the top level structure in UMACore. It is accessible from the Python REST API.

`Experiment:` An experiment object is equivalent to an `Experiment` in UMAClient. This object holds all `Agents` of the current experiment. Each instance can hold multiple `Agents`. It is accessible from the Python REST API.

`Agent:` A class implementing the functionalities of a BUA. A REST API is provided operating on agents directly. Currently, an Agent maintains two snapshots, implementing the functionalities of a *binary UMA agent* (BUA) defined in [1].

The Agent class contains several subclasses, corresponding to different types of UMA learners:

`Agent:` The `stationary` agent type. Every `Agent` class is by default of `stationary` type. This agent type, utilizes a discounted snapshot update, with automated control of learning thresholds.

`AgentDiscounted:` An agent type implementing the `discounted` update with fixed discount coefficient and learning thresholds. The`AgentDiscounted` inherits from `Agent` class.

`AgentEmpirical:` implements `empirical` UMA learning. This class, too, inherits from the `Agent` class.

`AgentQualitative:` implements a BUA based on `qualitative` (ranking-based) UMA learning. This class inherits from the `Agent` class.

Snapshot: implements a snapshot entity. A REST API is provided which operates on Snapshot objects directly. A Snapshot refers to its associated collections of Sensor, SensorPair, AttrSensor and AttrSensorPair objects (see below). It also refers to a DataManager object. Snapshot objects handle an overwhelming majority of service requests, but they do not handle the CPU/GPU data flow directly.

There are also four types of `Snapshot` corresponding to the available `Agent` types:

Snapshot: The parent class for all snapshot types. The`Snapshot` class is by default a `stationary` type. It is automatically added by `Agent`.

SnapshotDiscounted: Implementation of a snapshot of `discounted` type. The `SnapshotDiscounted` class inherits from the `Snapshot` class, and will be automatically added by `AgentDiscounted`.

SnapshotEmpirical: Implementation of an `empirical` snapshot. The `SnapshotEmpirical` class inherits from the `Snapshot` class, and will be automatically added by any `AgentEmpirical`.

SnapshotQualitative: Implements qualitative snapshots. The `SnapshotQualitative` class inherits from the `Snapshot` class, and will be automatically added by any `AgentQualitative`.

DataManager: this object type is essential for the data flow between GPU, CPU and all UMA objects. It is responsible for creating data arrays on both host and device, resizing those arrays, extracting them for computations, and cleaning them up. All low-level related data flow should use this object. For more info about the data flow in UMA, see Section 2.3.

Sensor: represents a sensor (Boolean query) entity assigned to the BUA (the parent Agent object), and it is accessible directly from the client side. It holds a 'positive', and a 'negative' `AttrSensor` object (corresponding to the actual query and its negation), and refers to variables stored in the BUA's `DataManager` object. During resizing operations, it is also responsible for storing some state variables.

SensorPair: Responsible for recording the information pertinent to learning and reasoning, such as the snapshot weights (see, e.g. [1] Figure 1) and learning thresholds. It controls four `AttrSensorPair` objects, and has pointers to the two `Sensor` objects making up the pair. During resizing, it is also responsible for storing the learning threshold value for the pair.

AttrSensor: This object stores attributes of the sensor from the vantage point of the parent Agent object. It stores the current observation (state of the sensor), and its current and preceding weights. It is accessible from REST API on client side.

AttrSensorPair: For every pair of `AttrSensor` objects, there is a `AttrSensorPair` object to manage them, which is controlled by the appropriate `SensorPair` object. An `AttrSensorPair` object stores the weight of an edge in the poc graph and the orientation of the edge (if any). It also refers to the `SensorPair` objects making up the pair.

Finally, a dedicated name space for all operations on the above objects is provided. See Section 2.5 for further details.

(6) `UMACoreTest(.exe)`

`UMACoreTest` is an executable containing unit test cases for the UMACore module. It depends on

UMAUtil, UMAKernel, and gtest.

(7) UMARest.so(.dll)

UMARest is another shared (dynamic linking) library, that only deals with REST functionality. It functions as a wrapper for cpprestsdk (Casablanca), to facilitate the construction of REST APIs without requiring knowledge of the 3rd party library. UMARest has three important object classes:

UMARestListener: Similarly to a server, its function is to catch input REST calls, and pass them to corresponding handlers.

UMARestHandler: This is the base class for all customized handlers. Upon receipt of a request from the listener, its task is to reply with a corresponding response.

UMARestRequest: An object of this class contains matching requests and response objects. It is tasked with holding all information about a request, and with converting input data into a format acceptable for cpprestsdk. It is also responsible for casting outgoing data into an appropriate type from the C++ std library, to facilitate communication with other UMA modeules.

(8) UMARestTest(.exe)

UMARestTest is an executable containing unit test cases for the UMARest module. It depends on UMAUtil and gtest.

(9) UMAc(.exe)

This is the final executable. When it launches, it will listen on a local port, and uses UMARest to serve incoming requests. When a request comes in, UMARest will pass it to the appropriate handler and call the desired functionality in UMACore.

(10) UMA(.exe)

UMA is a helper CLI executable that is used to manage the UMAc process. It supports start, stop and status options.

2.2.2. *C++ Exceptions.* The exception class resides in UMAUtil (which is included in all UMA modules), to ensure that any C++ exception raised by an execution of UMA can be handled in any of its modules. All Exceptions inherit from std::runtime_error, and are initialized with an error message, error level, and error type. Error messages are customizable.

Implemented error levels are: WARN, ERROR and FATAL. WARN means the exception being raised did not interrupt the current workflow, and it is rarely used in current code; ERROR means the workflow has been compromised; FATAL indicates the operation interrupted the UMA process, causing its immediate termination.

Error types classify the possible exceptions. When handling an exception, UMARest will return a status codes to the client, based on the error type. Particularly, UMA Core has the following type of Exceptions:

UMAInternalException: Any exception that is caused by an internal error, which is not expected in normal workflow. This type of exception will typically trigger FATAL condition.

UMAInvalidArgsException: This exception is likely to be triggered by an invalid input from REST call. Usually an ERROR condition will be triggered.

UMANoResourceException: When a REST call is querying an non-existing object, this exception should be thrown. Usually an ERROR condition will be triggered.

UMADuplicationException: When a REST call is trying to create an object with an exisiting `id`, this exception should be thrown. Usually an `ERROR` condition will be triggered.

UMABadOperationException: Thrown if a REST call is used illegally (e.g., wrong input format or illegal workflow). Usually an `ERROR` condition will be triggered, but a `FATAL` condition may also be triggered.

2.2.3. *C++ logging system.* Similarly to the exception handling system, the logging system is implemented in `UMAUtil` to ensure all modules can utilize it. Each architectural component maintains its own separate log, which can be set to a preferred level of detail: `ERROR`, `WARN`, `INFO`, `DEBUG`, `VERBOSE`. (If the log level is INFO, then message logged as DEBUG and VERBOSE will not be recorded.)

Log files are created when loggers are created. The logger will read from log.ini to get the component's log level. All loggers are static global variables.

2.2.4. *C++ initialization files.* Initialization files are located under `ini/` and are essential for the program to run. There are, so far, four such files.

`log.ini`: Sets the log level information for each log component.

`restmap.ini`: Contains the REST map information. A simple way to disable a REST endpoint is to comment(#) out the appropriate url.

`server.ini`: Contains the host and port information used to initialize the `UMARest` listener.

`core.ini`: Sets default parameters for UMA Core objects, such as the discount parameters and learning thresholds of Agent objects and the allocated memory expansion rate for Snapshot objects.

The content of a `.ini` file is organized by the architectural component relevant to the task. Each component's section of the file is headed by its name in square brackets. For each component, keys and key-value pairs are used to store its settings:

`log.ini`: Component names will match the list of Logger object names; 'level' is the only accepted key, which maps to the log level.

`restmap.ini`: Component names are the available REST handlers. Each component lists the URLs to be used for REST mapping.

`server.ini`: Has the server port and host message, under component 'Server'.

`core.ini`: Components are the names of UMA object classes. The key value pairs set default values that to be used by these objects during initialization.

The `core.ini` file influences some core parameters in subtle ways. In particular, it is important to understand the way in which initialization parameters are inherited/passed along among objects of the UMA hierarchy:

`UMA Core Objects:` UMA Core objects are objects representing (each) a single UMA identity. These include: `World`, `Experiment`, `Agent`, `Snapshot`, and `DataManager`.

`UMA Core Object` inheritance hierarchy: An `UMA Core Object` inherits properties from its ancestors based on three rules:

(1) All instances will inherit properties from their parents; the hierarchy from top to bottom is: `World`, `Experiment`, `Agent`, `Snapshot`, `DataManager`.

(2) All sub-classes will inherit the properties from their parent class. This applies to different types of `Agent` and `Snapshot`.

(3) Property input from `UMA Client` will assume the highest priority and override existing values in the current instance. However, values passed to the next level of the `UMA Core Object` hierarchy may still be overridden by the default value within that `UMA Core object`.

2.2.5. *An example core initialization file.* Let is consider the inheritance order of learning threshold values, as specified in a hypothetical `core.ini` file.

Overall, if a learning threshold value is provided in the `[World]` section of `core.ini`, that means the value belongs to the `World UMA Core Object`. As a result, all `UMA Core Objects` will be inheriting this value by default. To make this value only apply to all `Agent` objects, you need to provide them under the `Agent` section or under the `Agent::SubClass` section. If the value is provided under the `Agent` section, then every type of `Agent` will share this value, but if the value is defined under a specific type of `Agent`, then only this specific type will inherit the given value. Beyond this point, the threshold value may also be overridden by a run-time input.

Given the following hypothetical `core.ini` input,

- ```
  [World]
  [Experiment]
  [Agent]
  threshold=0.9
  [Agent::Qualitative]
  [Agent::Discounted]
  threshold=0.8
  [Snapshot]
  [Snapshot::Qualitative]
  threshold=0.7
  [Snapshot::Discounted]
  ```

the default `threshold` for all agents becomes `0.9`, with the exception of `Discounted` agents (`threshold=0.8`), while `Qualitative` snapshots end up sporting a `threshold` value of `0.7`. Furthermore, a run-time input updating the `threshold` of a `Discounted Agent` to `0.6` will result in updating the thresholds for its snapshots to `0.6` as well.

2.3. **UMA Data Flow.** The data flow in the UMA architecture is based on its separation into components presented in Figure 1.

On the Python side, the simulation engine will exchange data with the client APIs. The input from the simulation includes: current and preceding raw observation vectors for each BUA; current and preceding values of the motivational signal for each BUA; target specification vectors (for some BUAs) in some simulations; and, finally, BUA parameters. Output data are the decisions for each BUA; internal state vectors (target, prediction, current state,...) and results of batch-propagation.
  ```

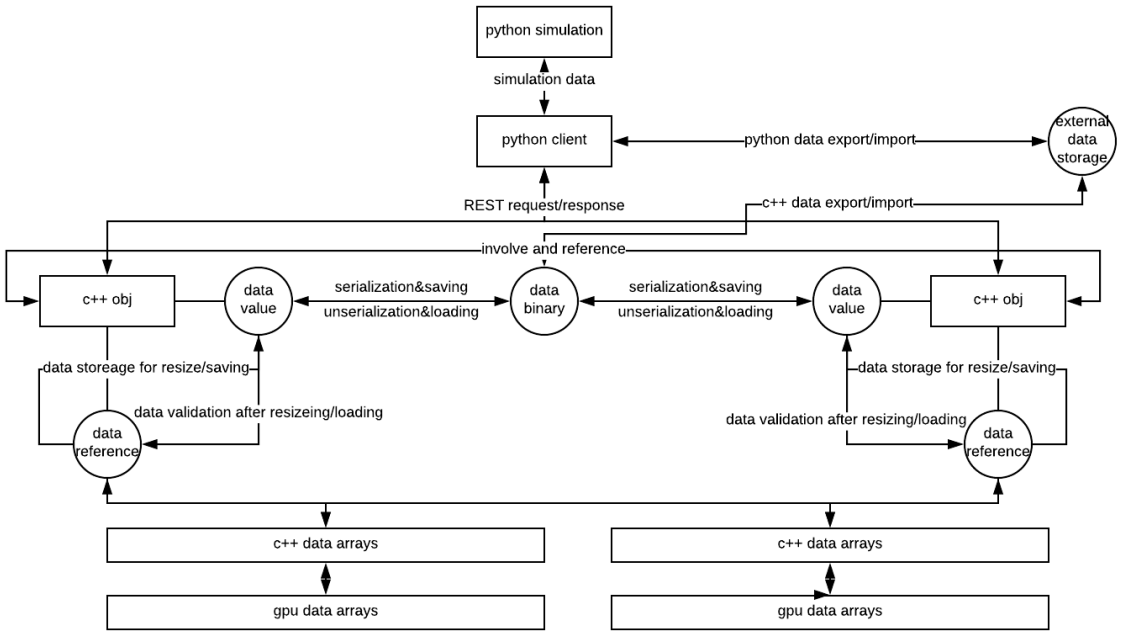

FIGURE 3. Data flow withing the C++ UMA core.

As a result, the Python client and UMA C++ Objects data flow is rather complex, having many data types to deal with in the context of different REST calls. Please check the corresponding REST API code for details.

Data flow between UMA C++ Objects and Host/Device Data is another important layer.

- To leverage parallel computation capabilities, storage space must be allocated in continuous regions (arrays) both on CPU and GPU, for every data element;
- Pointers to all relevant data arrays must be maintained;
- All the above are complicated by the enrichment operation available to BUAs: extra memory needs to be allocated to accommodate dynamic extension of all data structures, with periodic resizing (expansion) whenever the allocated space budget is used up; resizing is achieved by allocating a larger continuous region to each array (both on CPU and on GPU), copying the contents of the old array into this region and freeing the old region, which forces an update of all the relevant pointers.

2.4. **The UMA REST Workflow.** The UMA REST workflow represented in Figure 4 represents our implementation of `UMARest`. When a request is posted, `UMARest` will first locate the corresponding REST

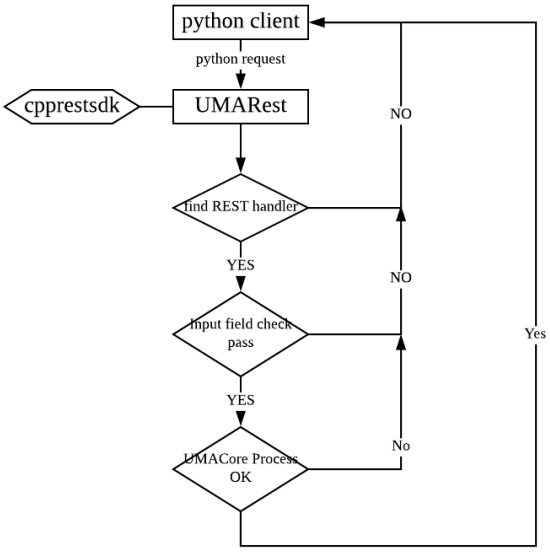

FIGURE 4. The UMA REST workflow.

handler. Then it will scan the input fields for data. Finally, it will call the appropriate module in `UMACore` to execute the appropriate back end operation. If everything is OK, `UMARest` will return a 200+ response to the client, along with output data. Otherwise, `UMARest` will return a 400+/500+ error flag with error messages.

2.5. **UMA REST Map.** The UMA REST map has seven handlers, all read from the `restmap.ini` file. The handlers are:

`WorldHandler:` Deals with agent management.

- `/UMA/world`: return the current world's agent info
- `/UMA/world/reset`: reset the world singleton object, will destroy it and recreate one.

`ExperimentHandler:` Deals with experiment management.

- `/UMA/object/experiment`: manage the CRUD in experiment.
- `/UMA/object/experiment/save:` manage the saving of an experiment.
- `/UMA/object/experiment/load:` manage the loading of an experiment.

`AgentHandler:` Manages Agent CRUD operations.

- `/UMA/object/agent`: CRUD Agent.
- `/UMA/object/agent/copy`: manage agent copy between experiments.

`SnapshotHandler:` Manages Snapshot CRUD and maintenance.

- `/UMA/object/snapshot`: manages CRUD in Snapshot.
- `/UMA/object/snapshot/init`: initializes a snapshot using all current sensors as initial sensors.
- `/UMA/snpashot/delay`: adds the specified delayed conjunctions to the snapshot.
- `/UMA/snapshot/amper`: adds the specified conjunctions to the snapshot.
- `/UMA/snapshot/pruning`: removes the specified delayed conjunctions from the snapshot.

`SensorHandler:` Manages Sensor/SensorPair CRUD.

- `/UMA/object/sensor`: CRUD sensor.
- `/UMA/object/sensor_pair`: CRUD sensor pair.

`AttrSensorHandler:` Manages `AttrSensor/AttrSensorPair` CRUD.

- `/UMA/object/attr_sensor`: CRUD `AttrSensor`.
- `/UMA/object/attr_sensor_pair`: CRUD `AttrSensorPair`.

`SimulationHandler:` Manages all simulation requests.

- `/UMA/simulation/decision`: launch another decision cycle.
- `/UMA/simulation/up`: perform upwards closure on a single input signal (deprecated).
- `/UMA/simulation/ups`: perform upwards closure on a batch of input signals.
- `/UMA/simulation/downs`: perform downwards closure on a batch of input signals.
- `/UMA/simulation/propagation`: perform propagation on a batch of input signals.
- `/UMA/simulation/npdirs`: enforce the calculation of the implications matrix based on the current poc-graph matrix.
- `/UMA/simulation/blocks`: perform single-linkage clustering at the specified scale on a batch of input signals.

- `/UMA/simulation/abduction`: perform abduction on a batch of input signals.
- `/UMA/simulation/propagateMasks`: enforce the propagation of the delay masks.

`DataHandler:` Manage all data arrays in core, for the host snapshot.

- `/UMA/data/observe`: update the raw observation signal.
- `/UMA/data/current`: update/read current state signal.
- `/UMA/data/prediction`: read prediction signal.
- `/UMA/data/target`: update/read target signal.
- `/UMA/data/dataSize`: read the current size info of all array variables.
- `/UMA/data/weights`: report the weights matrix.
- `/UMA/data/thresholds`: report the threshold matrix.
- `/UMA/data/dirs`: report the implications matrix.
- `/UMA/data/negligible`: report the indicator signal of the set of negligible sensors.
- `/UMA/data/propagateMasks`: report the implications matrix signal of the delay masks.
- `/UMA/data/maskAmper`: report the amper matrix signal of all existing sensors.
- `/UMA/data/all`: report all signals or array above in one call.

2.6. **UMA Test.** The project implements three kinds of tests: unit, functional, and performance tests.

The unit testing utility is built using `gtest`. To expand the collection of unit tests, add the test cases to the C++ test source, and run the test executables for the module.

To run performance tests, run `pylib/perf/resource.py` as a separate process to have system matrix usage recorded as an experiment is being run.

Functional testing is based on REST API. Construct your functional test cases in yml form, and run with `test_wrapper.py` (which is found under `pylib/client/`). The functional test checks the status code, message and data fields arriving in response to the implemented REST calls.

2.7. **Development Guide.** This section includes comments on our design considerations and how they should affect further development by the user.

2.7.1. *UMA Python Simulation Engine Development.* The simulation engine uses the UMA Python Client to interact with the UMA Core. Simulation scripts may have varying settings, but will share some common libraries (such as `som2_noEP.py`). Because it is preferable for the simulation scripts not to interact with with the UMA Python Client directly, we recommend keeping track of the capabilities of the Agent class of `som2_noEP.py` and expanding them as necessary.

In the common libraries, to interact with UMA Python Client, an `UMARestService` object is (and must be) created in the global scope. The `UMARestService` object is a wrapper class that handles all REST calls to the the UMA core, and is a part of the UMA Python Client. When an `UMARestService` is created, it does not maintain a connection to the UMA core: it only stores the necessary information (url, port, header,...) necessary for a REST call to successfully communicate with UMA core.

An `UMARestService` object is a required input to any UMA Python Client object, and we recommend that the service object already present in the global scope be used for this purpose.

Though it is possible to create `UMARestService` objects on demand, the resulting temporary objects are only available within their parent scope, and require significant memory resources for their storage. In addition, multiple active `UMARestService` objects may interfere with each other during logging operations, which might pose a significant challenge in a multi-threaded setting.

`UMARestService` is initialized with http scheme, host, and port designations. By default, it uses http, localhost and port 8000.

In addition to the `UMARestService` object, there are currently serveral python client classes that are used for REST calls:

`UMARestService:` is initialized with headers and basic url that can connect to an `UMA Core` instance;

`UMAClientObject:` is the base object of all UMA REST client object. It is initialized with an `UMARestService` instance;

`UMAClientWorld:` is only responsible for constructing Experiment objects;

`UMAClientExperiment:` is responsible for creating, updating or deleting Agent objects. It is initiated with an experiment id;

`UMAClientAgent:` is initialized with an experiment id and agent id. It is responsible for BUA-level maintenance operations (e.g. adding snapshots, managing snapshot parameters, making a decision based on the internal state of member snapshots).

`UMAClientSnapshot:` is initialized with an experiment id, an agent id (MID) and a snapshot id.[3]

The responsibilities of `Service_Snapshot` are quite diverse:

(1) Adding existing sensors (by MIDs) to the snapshot;

(2) Initializing snapshots (using the currently assigned sensors for initial sensors);

(3) execute the up/ups/downs/propagation/abduction/npdir operations;

[3]Either "plus" or "minus" in the current implementation.

(4) execute enrichment/pruning operations;

(5) compute, store and report internal state vectors (current/prediction/target/negligible. . .);

(6) set/update snapshot data in the UMA Core (q, threshold etc.).

`UMARestData:` is initialized with an experiment id, an agent id and a snapshot id. It is responsible for getting or setting the simulation data;

`UMARestSensor:` is initiated with an experiment id ,an agent id (MID), snapshot id (same as above) and a sensor ID. It is responsible for obtaining the delay mask (if any) of the current sensor in the UMA Core.

`UMAClientSimulation:` is initiated with an experiment id. It is responsible for doing all simulation related REST calls. The actual Experiment and Agent id used for simulation will be passed in the simulation function as a parameter.

To add a new REST functionality, define the corresponding UMA Python Client object (see below), and call the functions. Try to ensure all calls to the Python Client remain confined to `som2_noEP.py`.

2.7.2. *UMA Python Client Development.* UMA Python Client is the bridge between the UMA simulation engine and the UMA core. To add a new functionality in UMA Python Client, it is necessary to:

(1) Find/construct the REST endpoint in UMA Core;
(2) Design the input/output from the simulation engine.

All the functionalities currently implemented follow the following design format. First, the input to the new function, to be sent to the UMA core, is defined; as a rule, it is important that all input/output be shaped in terms of the Python base classes (e.g. lists, tuples). [4] Next, a CRUD REST request to the UMA Core (out of the options provided by the `UMA_service` instance, see Section 2.5) is made. Finally, we always check the return message within the function for error handling. If everything is OK, return the data to UMA Python Simulation.

2.7.3. *UMA Core Service Development.* UMA Core Service development has six parts (see Section 2.2 for a detailed review).

`UMAUtil`: This is the basic module of the UMA architecture. Any functionality introduced into this module must ensure its independence from 3rd party libraries. For cross platform functionalities (file creation, time function etc), we recommend the use of macros to distinguish between platforms, when necessary. All new functionalities in `UMAUtil` should come equipped with thorough tests: please add new test cases in `UMAUtilTest`.

`UMAKernel`: This is the module containing all kernel code running on CUDA. It should only depend on `cuda` and `UMAUtil`. When adding a new source file to this package, be sure to change the build script on Linux, so that the device code is built and linked correctly.

`UMACore`: This module defines all UMA objects. When making changes to existing functionalities or adding new functionalities, ensure those changes are consistent with the content of the corresponding objects in the UMA Python simulation engine.

`UMARest`: The module providing the UMA Core with REST functionality. It encapsulates the library `cpprestsdk` within itself. Therefore, when making alterations to this module, one must ensure all inputs/outputs of interface functions use only types provided in `std`. This is important because it can efficiently reduce 3rd party library dependencies on other modules, saving memory and, most critically, computation time.

`UMAc`: The executable part of UMA Core Service. When a new REST handler or new REST mapping is needed, just add a new handler class or new functions to handle the new REST URL.

`UMA`: The helper CLI wrapper around the `UMAc` module. It supports `start`, `stop`, `status` options which will help manage the UMAc process.

References

[1] D.P. Guralnik and D.E. Koditschek. The UMA-SOM Platform for General Learning (Technical Report). *technical report*, 2017.

[4] Generically, the UMA core should only be sent floating point values (such as the motivational signal or simulation parameters) and Boolean vectors/matrices (sensory signals).