APPENDIX A

Prototype Documentation

This appendix provides a documentation of our platform *Runtime and Analytics for Hybrid Computing Systems* (RAHYMS). This platform is a prototype of our proposed architecture, models, and frameworks. The platform serves as a proof-of-concept to showcase a realization of quality-aware and reliable Hybrid Human-Machine Computing Systems (HCSs).

In this appendix, first we describe how to get started with the platform. Afterwards, we discuss details of the operation of the platform in the simulation-mode, i.e., how to configure the simulation, and in the interactive-mode, i.e., how to use the Application Programming Interfaces (APIs).

A.1 Getting Started

A.1.1 Overview

This prototype is available as an open-source project, and can be cloned from a GitHub repository `https://github.com/tuwiendsg/RAHYMS`.

The project is developed using Java SDK 1.7, and can be built and deployed using maven. The root project is named *hcu* (stands for *hybrid compute units*). The *hcu* project contains the following sub-projects in their respective directories:

- `hcu-cloud-manager` contains a component to manage compute units including their functional capabilities and non-functional properties, a generator to populate the pool of compute units and to generate task requests in simulation-mode, and a compute unit discovery service. This sub-project also contains tool for calculating the reliability of the individual compute units as well as compute units collectives based on the property of the managed compute units.

- `hcu-common` contains utilities required by the platform, such as the models for compute units and tasks, interfaces to connect different component in the HCS, fuzzy libraries, configuration reader utilities, and tracer utilities.

- `hcu-composer` contains models and library for the formation engine to provision the compute units collectives.

- `hcu-external-lib` contains some adapted external libraries, i.e., GridSim and JSON for Java.

- `hcu-monitor` contains the code for the monitoring framework of HCS. It contains utilities, e.g., to create and deploy monitoring agents, to define, publish, and subscribe metrics, and also a Drools-based rule engine.

- `hcu-rest` contains a Jetty-based Web server for running the interactive-mode. It creates three HTTP services: one service runs the REST API server, one service provides the Web user interface, and another one provides a REST API playground developed using Swagger.

- `hcu-simulation` contains code for running a simulation using GridSim framework.

Additionally, the `smartcom` project is also available in the root as a tool for virtualizing communication with compute units. This project is adopted from the `smartcom` repository available online [1].

A.1.2 Building

For each root project and sub-projects, a maven configuration is provided to allow easy building and importing to an IDE for Java language. Before building the `hcu` project, we first need to build the required `smartcom` project. To build everything, run the following maven commands from the root directory of the repository:

```
$ cd smartcom
$ mvn install
$ cd ../hcu
$ mvn install
```

The jar files should now have been created by maven in each projects under the `target` directories. Particularly, two jar files
`hcu/hcu-simulation/target/hcu-simulation-0.0.1-SNAPSHOT.jar`, and
`hcu/hcu-rest/target/hcu-rest-0.0.1-SNAPSHOT.jar`
contain main classes for running the program in simulation- and interactive-mode respectively.

[1] `https://github.com/tuwiendsg/SmartCom`

A.2 Simulation Mode

To run the program in simulation-mode, from the root of the repository simply execute

```
$ java -jar hcu/hcu-simulation/target/hcu-simulation-0.0.1-SNAPSHOT.jar <config-file>
```

where the *<config-file>* argument is the path of the main configuration file.

To execute the program within an IDE, run the main class
at.ac.tuwien.dsg.hcu.simulation.RunSimulation
inside the hcu-simulation project with the *<config-file>* as the execution argument.

The main simulation configuration file *<config-file>* is a java properties file containing references to other configuration files specifying a simulation scenario, a composer (i.e., the formation engine) configuration, a tracer configuration, and a monitoring configuration. Listing A.1 shows an example of the main simulation configuration.

```
scenario_config = scenarios/samples/infrastucture-maintenance/scenario.json
composer_config = config/composer.properties
tracer_config   = config/tracer.json
monitor_config  = config/monitor.json # optional
```

Listing A.1: An Example of Main Simulation Configuration

We discuss the content of each configuration as follows.

A.2.1 Scenario Configuration

Our simulation of an HCS consists of two phases:

i) *Initiation Phase* is a phase where compute units are generated with configurable initial properties.

ii) *Execution Phase* is a phase where tasks are generated, and for each task a compute units collective is created to execute the task. The execution phase consists of cycles. In every cycle, the task generator configurations are processed to generate tasks. After a configured number of cycles have passed, the task generation stops, and simulation is finished once all the remaining running tasks are completed.

A simulation scenario mainly has two purposes: it defines how the compute units are generated during the initiation phase, and it defines the generation of task requests during the execution phase. A configuration of a simulation scenario is a json file. An example of a simulation scenario configuration is shown in Listing A.2.

```
{
  "title":"Infrastucture Breakdown Sensing",
  "numberOfCycles":100,
  "waitBetweenCycle":1,     ▶ delay (in simulation time unit) between each cycle
  "service_generator":{
    "basedir":"service-generator/",
    "files":[
      "inspector-generator.json",
      "citizen-generator.json",
```

```
      "sensor-generator.json"
    ]
  },
  "task_generator":{
    "basedir":"task-generator/",
    "files":[
      "machine-sensing-task-generator.json",
      "human-sensing-task-generator.json",
      "mixed-sensing-task-generator.json"
    ]
  }
}
```

Listing A.2: An Example of Scenario Configuration

Compute Units Generator Configuration

The `service_generator` element in the simulation configuration defines the list configuration for generating compute units together with their provided services (i.e., functional capabilities) and their properties. The `basedir` specifies the directory in which the compute units generator locates the specified `files` list. In Listing A.3, we exemplify a compute units generator configuration annotated to describe the purpose of the configuration. This example shows a generation of citizens as compute units.

```
{
  "seed":1001,     ▶ random number generator seed
  "numberOfElements":200,     ▶ number of compute units generated
  "namePrefix":"Citizen",
  "connection":{
    "probabilityToConnect":0.4,   ▶ probabiity of a compute unit connected to others
    "weight":<distribution-config>
  },
  "services":[     ▶ the functional services provided by each generated compute unit
    {
      "functionality":"DataCollection",
      "probabilityToHave":0.7,     ▶ probability the compute unit has this
          functionality
      "properties":[    ▶ functionality-specific properties
        <property-config>,
        ...
      ]
    },
    ...
  ],
  "commonProperties":[    ▶ non-functional properties
    <property-config>,
    ...
  ]
}
```

Listing A.3: An Example of Compute Units Generator Configuration

The *<distribution-config>* defines how a value should be populated with a random number generator, while the *<property-config>* specifies how each property is defined. They are defined in Listing A.5 and Listing A.5 respectively.

```
<distribution-config> ::=
```

```
{
  "class":"<distribution-class-name>",
  "params":[...],
  "sampleMethod":"..."
  "mapping":{    ▶ optional
    "0":"<mapped-value-0>",
    "1":"<mapped-value-1>",
    ...
  }
}
```

Listing A.4: Distribution Configuration

The *<distribution-class-name>* is the random number generator class which will be used to generate the random values. It can be any of distribution classes available from Apache Common Math package `org.apache.commons.math3.distribution`[2]. Other distribution classes can also be used by specifying a fully-classified class name. The `params` entry specifies the parameters required the instantiate the distribution class, for example, `NormalDistribution` class can be instantiated using a constructor with three numbers, e.g., $[0.30, 0.10, 1.0E-9]$, which define mean, standard deviation and inverse cumulative distribution accuracy respectively. The `sampleMethod` is a zero-argument method that should be invoked for getting the random values, the default is `sample` for the Apache Common's distribution classes. The optional `mapping` entry defines a mapping from an integer number distribution to a certain value, e.g., a string value.

```
<property-config> ::=
{
  "name":"<property-name>",   ▶ e.g., "location", "cost", etc.
  "probabilityToHave":1.0,   ▶ probability the compute unit has this property
  "type":"<property-type>",   ▶ can be "metric", "skill", or "static"
  "value":<distribution-config>    ▶ required for other than "metric" types
  "interfaceClass":"<property-type>"    ▶ required for "metric" type
}
```

Listing A.5: Property Configuration

A property can be of three types: `metric` property, which defines a property whose value can be retrieved externally, `skill` property, which defines the functional capability of a human-based compute units, and `static` is for all other properties (note that despite of the name, the `static` property value can still be modified by calling the property's setter method during runtime). For `metric` property, the `interfaceClass` entry defines the class implementing `MetricInterface` that provides the value of the property.

Task Generator Configuration

The `task_generator` element in the simulation configuration defines the list configuration for generating tasks at each cycle during the execution phase. The way how

[2] https://commons.apache.org/proper/commons-math/apidocs/org/apache/commons/math3/distribution/package-summary.html

basedir and files list work is the same as in the compute units generator configuration. Listing A.6 exemplifies an annotated task generator configuration.

```
{
    "seed": 1001,     ▶ random number generator seed
    "taskTypes": [    ▶ list of task types that should be generated
        {
            "name": "HumanSensingTask",
            "description": "An explanation of the task",
            "tasksOccurance": {*\textit{<distribution-config>}*},   ▶ number of tasks generated at each cycle
            "load": {<distribution-config>}, ▶ to simulate how long the task will be executed by a unit
            "roles": [  ▶ list of roles for the task
                {
                    "functionality": "DataCollection",   ▶ a functional requirement for the role
                    "probabilityToHave": 1.0,    ▶ probability the role has this functional requirement
                    "relativeLoadRatio": 1.0,    ▶ effective load = relative load *task load
                    "dependsOn": ["...", ...],   ▶ a list of role functionality that this role depends on (collective dependency)
                    "specification": [  ▶ role-level non-functional constraints
                        <specification-config>,
                        ...
                    ]
                },
                ...
            ],
            "specification": [  ▶ task-level non-functional constraints
                <specification-config>,
                ...
            ]
        },
        ...  ▶ multiple task types can be defined
    ]
}
```

Listing A.6: An Example of Task Generator Configuration

The *<distribution-config>* is similar to the one used in the compute units generator configuration. The *<specification-config>* defines non-functional constraints as specified in Listing A.7.

```
<specification-config> ::=
{
  "name":"<property-name>",   ▶ e.g., "location", "cost", etc.
  "probabilityToHave":1.0,    ▶ probability the requirement has this constraint
  "type":"<property-type>",   ▶ can be "metric", "skill", or "static"
  "value":"<distribution-config>",
  "comparator": "<comparator-class>"
}
```

Listing A.7: A Specification of Non-Functional Constraints

The *<comparator-class>* is a fully-qualified name of a class implementing the `java.util.Comparator` interface. Several comparator classes are provided in `at.ac.tuwien.dsg.hcu.common.sla.comparator` package: `StringComparator`, `NumericAscendingComparator`, `NumericDescendingComparator`, and `FuzzyComparator`.

A.2.2 Formation Engine Configuration

A formation engine configuration is a java properties file specifying the algorithm used by the formation engine, and the parameters required by the algorithms. Listing A.8 shows a snippet example of composer configuration. Currently, available formation algorithms are

- `FairDistribution` algorithm, which distributes tasks uniformly to all qualified compute units,
- `PriorityDistribution` algorithm, which distributes tasks based on the priority of each compute unit specified in `assignment_priority` property, e.g., a compute unit with priority equals to 2 has twice probability to be assigned to tasks compared to compute units with priority equals to 1,
- `EarliestResponse` algorithm, which assigns tasks to compute units with the earliest estimated response time (e.g., the *first come first serve* strategy),
- `GreedyBestFitness` algorithm is a greedy heuristic strategy, which processes each task role iteratively, and for each role a compute unit with the best local fitness value is selected,
- `GreedyHillClimbing` algorithm finds an initial solution similarly as the `Greedy BestFitness` algorithm, and refines the solution further using a *hill climbing* technique, the number of cycles for hill climbing is specified using `maximum_number_of_cycles` parameter,
- `ACOAlgorithm` algorithms, which find the best solution using Ant Colony Optimization. Currently the following variants of ACO algorithms are supported: `AntSystemAlgorithm`, `MinMaxAntSystemAlgorithm`, and `AntColonySystemAlgorithm`. ACO algorithms have many configurable parameters. An example of complete composer configuration including all the parameters can be found in `config/composer.properties` inside the `hcu-composer` project.

```
algorithm = ACOAlgorithm
aco_variant = AntSystemAlgorithm
#aco_variant = MinMaxAntSystemAlgorithm
#aco_variant = AntColonySystemAlgorithm

#algorithm = FairDistribution
#algorithm = PriorityDistribution
#algorithm = EarliestResponse
#algorithm = GreedyBestVisibility
#algorithm = GreedyLocalSearch
```

Listing A.8: An Example of Formation Engine Configuration

A.2.3 Tracer Configuration

The tracer configuration is a json file, which specifies the location of trace files (in CSV format) generated during runtime. There are two default tracers, named `reliability` and `composer` tracers, which are used by the formation engine and reliability analysis engine, respectively, for generating the traces of compute units collectives formation created and the reliability measurement for each task execution.

```
[
    {
        "name": "composer",
        "file_prefix": "traces/composer/composer-sample-",
        "class": "at.ac.tuwien.dsg.hcu.composer.ComposerTracer"
    },
    {
        "name": "reliability",
        "file_prefix": "traces/reliability/reliability-sample-",
        "class": "at.ac.tuwien.dsg.hcu.cloud.metric.helper.ReliabilityTracer"
    }
]
```

Listing A.9: An Example of Tracer Configuration

A custom tracer can be created by creating a new class extending `at.ac.tuwien.dsg.hcu.util.Tracer`, and adding a new corresponding entry in the tracer configuration. The new tracer can be invoked anywhere within the program by calling `Tracer.getTracer("<tracer-name>")`.

A.3 Interactive Mode

The following command can be executed to run the program in interactive-mode:

```
$ java -jar hcu/hcu-rest/target/hcu-rest-0.0.1-SNAPSHOT.jar <config-file>
```

where the `<config-file>` argument is the path of the main configuration file.

Within an IDE, the interactive-mode can be started by running the main class `at.ac.tuwien.dsg.hcu.rest.RunRestServer` inside the `hcu-rest` project with the `<config-file>` as the execution argument.

The main configuration file for interactive-mode contains HTTP server configuration, as well as the composer configuration for the formation engine. Note that currently we do not yet support monitoring and reliability analysis in interactive-mode.

Listing A.10 shows an example of configuration for interactive mode. The formation engine configuration defined in `composer_config` has the same format as the `composer_config` in the simulation-mode.

```
SERVER_PORT = 8080
SERVER_HOST = localhost
REST_CONTEXT_PATH = rest
WEBUI_CONTEXT_PATH = web-ui
SWAGGER_CONTEXT_PATH = rest-ui

composer_config = config/composer.properties
```

Listing A.10: An Example of Configuration for Interactive-Mode

Once, the program is started in interactive-mode, the Jetty-based HTTP server is started and listening on the port specified in the configuration. Afterwards, the services can be accessed from

- `http://<SERVER_HOST>:<SERVER_PORT>/<REST_CONTEXT_PATH>` for the RESTful *Application Programming Interfaces* (APIs),
- `http://<SERVER_HOST>:<SERVER_PORT>/<WEBUI_CONTEXT_PATH>` for the *Web User Interface*, and additionally
- `http://<SERVER_HOST>:<SERVER_PORT>/<SWAGGER_CONTEXT_PATH>` for the REST *API playground* based on Swagger.

Note that current prototype implementation of the interactive-mode does not expose full capabilities of the underlying models and framework as found in the simulation-mode. We discuss the APIs provided by our platform as follows.

A.3.1 Application Programming Interface

The Application Programming Interfaces (API) provided by the platform is a RESTful API, which provides CRUD (create, read, update, delete) operations on four entities: `unit`, `task`, `collective`, and `task_rule`.

Below is a list of applicable information for all APIs:

Request URL prefix
: `http://<SERVER_HOST>:<SERVER_PORT>/<REST_CONTEXT_PATH>/api`
default: `http://localhost:8080/rest/api`

`POST` and `PUT` parameters encoding (in the request body)
: `application/x-www-form-urlencoded`

HTTP response codes
: `200`: Successful
`201`: Created successfully
`404`: Error, entity not found
`409`: Error, entity already exists

Response body encoding
: `application/json`

The `unit` and `task` entities and their API operations are described as follows. Documentation of API operations for other entities can be viewed online from the Swagger API playground provided by the platform. When an API excepts a URL parameter, it is shown here inside curly brackets. Actual request should not include the brackets in the URL.

Operations on `unit`

a) **`GET /unit`** ► *List all units*
Response body on success:

```
[
  {
    "name": "...",
    "email": "...",
    "rest": "...",
    "services": [
      "...", ...
    ],
    ...
  },
  ...
]
```

Note:
– `rest` is the REST service URL for software-based compute units
– `services` is a list of functional capabilities provided by the compute units

b) **`GET /unit/{email}`** ► *Find a unit by email*
Response body on success:

```
{
  "name": "...",
  "email": "...",
  "rest": "...",
  "services": [
  "...", ...
  ],
  "elementId": 1
}
```

Note: refer to note for `GET /unit`

c) **`POST /unit`** ► *Create a new unit*
Parameters:
– `email` (string)
– `name` (string)
– `rest` (string, optional)
– `services_provided` (string): a comma separated string containing a list of functional capabilities provided by the compute unit, e.g., "`DataCollection, DataAssessment`".

d) **`PUT /unit/{email}`** ► *Update an existing unit specified by email*
Parameters:
– `email` (string)
– `name` (string, optional)
– `rest` (string, optional)
– `services_provided` (string): a comma separated string containing a list of functional capabilities provided by the compute unit
Response body on success: refer to response body for `POST /unit`

e) **`DELETE /unit/{email}`** ▶ *Delete an existing unit specified by email*
Response body on success: None

Operations on `task`

In this API, we simplify the task entity model. Each task request has `tag` (e.g., a category) and `severity` (e.g., `'NOTICE'`, `'WARNING'`, `'CRITICAL'`, `'ALERT'`, or `'EMERGENCY'`) properties. When the task request is processed, it is expanded using `task_rule` to a more complete task specification containing the functional capabilities (i.e., services) required to execute the tasks. Here, one service corresponds to one task role. Currently, we do not support updating and deleting a task, because the task is immediately assigned to and executed by the provisioned compute units collectives.

a) **`GET /task`** ▶ *List all tasks*
Response body on success:

```
[
  {
    "id": 1,
    "name": "...",
    "content": "...",
    "severity": "...",
    "tag": "...",
    "timeCreated": "...",
    "collectiveId": 1
  },
  ...
]
```

Note:
- `id` is an auto-generated id of the task
- `collectiveId` is the id of the compute units collective provisioned to execute the task

b) **`GET /task/{id}`** ▶ *Find a task by id*
Response body on success:

```
{
  "id": 1,
  "name": "...",
  "content": "...",
  "severity": "...",
  "tag": "...",
  "timeCreated": "...",
  "collectiveId": 1
}
```

Note: refer to note for `GET /task`

c) **`POST /task`** ▶ *Submit a new task request*
Parameters:
- `name` (string): Task's name
- `content` (string): Task's content description
- `tag` (string): Task's tag, e.g., a category

– `severity` (SeverityLevel) = [`'NOTICE'`, `'WARNING'`, `'CRITICAL'`, `'ALERT'`, or `'EMERGENCY'`]: Task's severity