Language Framework for Optimal Schedulers (LFOS)

Guideline for Users

Güner Orhan

December 23, 2016

1 Required Modules

In order to use LFOS frameowrk API, a programmer should import the required module:

```
from LFOS.Scheduler.Scheduler import Scheduler
from LFOS.Resource.Resource import *
from LFOS.Task.Task import *
from LFOS.Scheduling.Characteristic.Time import Time
from LFOS.macros import *
```

Listing 1: Importing required modules

2 Scheduler

Based on the selected instance, the scheduler instance, namely "sched", can be generated by following the following procedures one-by-one:

2.1 Resource Initialization for "cpu1"

Since some of the task specifications are based on the resources, in the framework, a programmer is expected to define the resources, initially. As explained in the article, some of the specifications are inevitable for a resource. Therefore, it should be defined for each resource. Based on the feature model, the following attributes are inevitable for a resource:

- Capacity ($\mathcal{C}$): The capacity of the resource are required to determine the maximum amount of capacity which can be utilized per time unit.
- Type ($\Re$): This attribute categorize the resources based on *Abstractions* and *Identifier*.
- Mode ($\mathcal{X}$): The mode of a resource may be either *Shared* or *Exclusive*.

- POWER CONSUMPTION ($\mathcal{V}$): A resource consumes power based on this attribute. The resource is either *Scalable* or not.
- OBJECTIVE ($\mathcal{O}_\alpha$): This attribute is related with the resource-related objectives.

The only required specification for the instantiation is *Type* of the resource. The default values for all specification belonging to a resource are shown in Table 1.

Feature Name	**Variable Type**	**Initial Value**
Capacity ($\mathcal{C}$)	float	0.0
Type ($\Re$)	LFOS.Resource.Type.ResourceTypeList::Enum	proc_t
Mode ($\mathcal{X}$)	LFOS.Resource.Mode.ModeTypeList::Enum	CB_EXCLUSIVE
Power Consumption ($\mathcal{V}$)	LFOS.Resource.Power	None
Objective ($\mathcal{O}_\alpha$)	LFOS.Objective	None

Table 1: Default instance variables of the *Resource* module and their default values.

For *abstraction*, ACTIVE is selected. Therefore, you can create the type object using the following code segment:

```
cpu1_t = Type(ResourceTypeList.ACTIVE, 'cpu1_t')
```

Listing 2: Active resource type object instantiation

According to the specification, a programmer can create the resource giving type object and a name of the resource as arguments to the class method of the ResourceFactory class shown in Listing 3. For active resources, an object belonging to TerminalResource class is instantiated using factory method pattern to handle the optional feature under *Abstraction* sub-feature.

```
cpu1 = ResourceFactory.create_instance(cpu1_t, 'cpu1')
```

Listing 3: Active resource instantiation using ResourceFactory class

2.1.1 Setting mode

There are three possible types for this attribute. These are:

- ModeTypeList.SHARED
- ModeTypeList.CB_EXCLUSIVE
- ModeTypeList.SB_EXCLUSIVE
- ModeTypeList.CB_AND_SB_EXCLUSIVE

The functionality of these modes are discussed in the article.

As shown in Table 1, the mode is initially set to ModeTypeList.CB_EXCLUSIVE. A programmer can change the mode of a resource by using the following code segment:

```
cpu1.set_mode(mode)
# mode --> ResourceTypeList::Enum
```

Listing 4: Setting the mode of a resource after creating a resource.

In order to check the mode of the resource, you can use the functions depicted in Listing 5.

```
cpu1.is_mode(mode) # mode --> ResourceTypeList::Enum
# returns True if the argument matches with the mode of the resource.

cpu1.is_exclusive()
# returns True if the mode of the resource is any one of the exclusive mode
    .
```

Listing 5: The functions for resource mode check.

According to your specification, you have selected at least CB_EXCLUSIVE mode for your resource "cpu1". Since the resource is set to this mode initially, you do not need to set it again.

In addition to the CB_EXCLUSIVE mode, you have selected the SB_EXCLUSIVE mode for your resource "cpu1". Therefore, you should manually set it after resource creation using the following code segment:

```
cpu1.set_mode(ModeTypeList.CB_AND_SB_EXCLUSIVE)
```

Listing 6: The resource is set to CB_AND_SB_EXCLUSIVE mode.

Due to semantic-based exclusive property of the resource, you can define exclusive resources by using the following formula:

```
cpu1.add_exclusive_resource(resource)
# returns True if the SB\_EXCLUSIVE mode is selected and the resource
    argument is not in the list of exclusive resources. Otherwise, it
    returns False.
```

Listing 7: A function for adding exclusive resources.

2.1.2 Setting Power Consumption

There are three possible types of power consumption:

- PowerTypeList.FIXED_STATE_POWER_CONSUMPTION
- PowerTypeList.DISCRETE_STATE_POWER_CONSUMPTION
- PowerTypeList.CONTINUOUS_STATE_POWER_CONSUMPTION

Each of these types have their corresponding classes inheriting Resource class. Therefore, we have utilized factory method design pattern.

You have selected continuous-state power consumption. Therefore, you can create your power consumption object with the code as follows:

```
power_type = PowerTypeList.CONTINUOUS_STATE_POWER_CONSUMPTION
cpu1_pc = PowerFactory.create_instance(power_type, scale, consumption)
# create_instance(_type, min_scale, min_pow_cons, max_scale=None,
    max_pow_cons=None) -> FixedStatePowerConsumption |
    DiscreteStatePowerConsumption | ContinuousStatePowerConsumption
#
#         Returns the corresponding instance for given _type.
#
#         :param _type:
#             PowerTypeList.FIXED_STATE_POWER_CONSUMPTION |
#             PowerTypeList.DISCRETE_STATE_POWER_CONSUMPTION |
#             PowerTypeList.CONTINUOUS_STATE_POWER_CONSUMPTION
#         :param min_scale: float -> minimum power scale
#         :param min_pow_cons: float -> minimum power consumption
#         :param max_scale: float -> maximum power scale
#         :param max_pow_cons: float -> maximum power consumption
#         :return: FixedStatePowerConsumption |
    DiscreteStatePowerConsumption | ContinuousStatePowerConsumption
```

Listing 8: Power consumption object is created for Continuous-State Power Consumption type.

All other member functions for the class is shown in Listing 9

```
ContinuousStatePowerConsumption.set_max_state(scale, pow_cons) -> boolean
ContinuousStatePowerConsumption.max_range_check(scale) -> boolean
ContinuousStatePowerConsumption.get_power_consumption_w_scale(scale) ->
    float
ContinuousStatePowerConsumption.get_active_power_state() -> dict
ContinuousStatePowerConsumption.get_max_power_state() -> list
ContinuousStatePowerConsumption.get_min_power_state() -> list
ContinuousStatePowerConsumption.get_power_states() -> Numpy.array
ContinuousStatePowerConsumption.set_min_state(scale, pow_cons) -> boolean
ContinuousStatePowerConsumption.remove_state(scale) -> boolean (Interface
    Function)
ContinuousStatePowerConsumption.range_check(scale) -> boolean
ContinuousStatePowerConsumption.get_power_consumption() -> float
ContinuousStatePowerConsumption.set_power_scale_precision(precision) ->
    None
ContinuousStatePowerConsumption.set_power_mode(scale) -> list(scale, power
    consumption)
ContinuousStatePowerConsumption.get_power_scale_precision() -> float
ContinuousStatePowerConsumption.__calculate_power_consumption_slope()
ContinuousStatePowerConsumption.add_state(self, scale, pow_cons) -> boolean
     (Interface Function)
ContinuousStatePowerConsumption.get_power_scale() -> float
```

Listing 9: The member functions for ContinuousStatePowerConsumption module.

2.2 Resource Initialization for "cpu2"

For *abstraction*, ACTIVE is selected. Therefore, you can create the type object using the following code segment:

```
cpu2_t = Type(ResourceTypeList.ACTIVE, 'cpu2_t')
```

Listing 10: Active resource type object instantiation

According to the specification, a programmer can create the resource giving type object and a name of the resource as arguments to the class method of the ResourceFactory class shown in Listing 11. For active resources, an object belonging to TerminalResource class is instantiated using factory method pattern to handle the optional feature under *Abstraction* sub-feature.

```
cpu2 = ResourceFactory.create_instance(cpu2_t, 'cpu2')
```

Listing 11: Active resource instantiation using ResourceFactory class

2.2.1 Setting mode

There are three possible types for this attribute. These are:

- ModeTypeList.SHARED
- ModeTypeList.CB_EXCLUSIVE
- ModeTypeList.SB_EXCLUSIVE
- ModeTypeList.CB_AND_SB_EXCLUSIVE

The functionality of these modes are discussed in the article.

As shown in Table 1, the mode is initially set to ModeTypeList.CB_EXCLUSIVE. A programmer can change the mode of a resource by using the following code segment:

```
cpu2.set_mode(mode)
# mode --> ResourceTypeList::Enum
```

Listing 12: Setting the mode of a resource after creating a resource.

In order to check the mode of the resource, you can use the functions depicted in Listing 13.

```
cpu2.is_mode(mode) # mode --> ResourceTypeList::Enum
# returns True if the argument matches with the mode of the resource.

cpu2.is_exclusive()
# returns True if the mode of the resource is any one of the exclusive mode
    .
```

Listing 13: The functions for resource mode check.

According to your specification, you have selected at least CB_EXCLUSIVE mode for your resource "cpu2". Since the resource is set to this mode initially, you do not need to set it again.

2.2.2 Setting Power Consumption

There are three possible types of power consumption:

- PowerTypeList.FIXED_STATE_POWER_CONSUMPTION
- PowerTypeList.DISCRETE_STATE_POWER_CONSUMPTION
- PowerTypeList.CONTINUOUS_STATE_POWER_CONSUMPTION

Each of these types have their corresponding classes inheriting Resource class. Therefore, we have utilized factory method design pattern.

You have selected discrete-state power consumption. Therefore, you can create your power consumption object with the code as follows:

```
power_type = PowerTypeList.DISCRETE_STATE_POWER_CONSUMPTION
cpu2_pc = PowerFactory.create_instance(power_type, scale, consumption)
# create_instance(_type, min_scale, min_pow_cons, max_scale=None,
    max_pow_cons=None) -> FixedStatePowerConsumption |
    DiscreteStatePowerConsumption | ContinuousStatePowerConsumption
#
#         Returns the corresponding instance for given _type.
#
#         :param _type:
#             PowerTypeList.FIXED_STATE_POWER_CONSUMPTION |
#             PowerTypeList.DISCRETE_STATE_POWER_CONSUMPTION |
#             PowerTypeList.CONTINUOUS_STATE_POWER_CONSUMPTION
#         :param min_scale: float -> minimum power scale
#         :param min_pow_cons: float -> minimum power consumption
#         :param max_scale: float -> maximum power scale
#         :param max_pow_cons: float -> maximum power consumption
#         :return: FixedStatePowerConsumption |
    DiscreteStatePowerConsumption | ContinuousStatePowerConsumption
```

Listing 14: Power consumption object is created for Discrete-State Power Consumption type.

All the other member functions for the object is shown in Listing 15:

```
DiscreteStatePowerConsumption.get_power_states() -> Numpy.array
DiscreteStatePowerConsumption.set_power_mode(scale) -> list
DiscreteStatePowerConsumption.range_check(scale) -> boolean
DiscreteStatePowerConsumption.get_max_power_state() -> list
DiscreteStatePowerConsumption.get_min_power_state() -> list
DiscreteStatePowerConsumption.get_active_power_state() -> dict
DiscreteStatePowerConsumption.remove_state(scale) -> float | None
DiscreteStatePowerConsumption.max_range_check(scale) -> boolean
DiscreteStatePowerConsumption.get_power_consumption_w_scale(scale) -> float
DiscreteStatePowerConsumption.add_state(scale, pow_cons) -> boolean
DiscreteStatePowerConsumption.set_max_state(scale, pow_cons) -> boolean
DiscreteStatePowerConsumption.get_power_scale() -> float
DiscreteStatePowerConsumption.get_power_consumption() -> float
DiscreteStatePowerConsumption.set_min_state(scale, pow_cons) -> boolean
```

Listing 15: The member functions for DiscreteStatePowerConsumption module.

Keyword	**Type**	**Default Value**	**Description**
name	string	-	The name of a task
periodicity	LFOS.Task.Periodicity.PeriodicityTypeList	-	The periodicty type of a task
deadline_type	LFOS.Task.Requirement.DeadlineRequirementTypeList	DeadlineRequirementTypeList.HARD	The deadline satisfaction type of a task
priority	int	0	The priority of a task
deadline	LFOS.Scheduling.Characteristic.Time	-	The deadline of a task
token_number	list(int)	[1]	The list of numbers of a specific token that would be fired. There exist one-to-one relation between the list and token_name list with respect to index.
phase	LFOS.Scheduling.Characteristic.Time	-	The first release time of a task
preemptability	LFOS.Task.Preemptability.PreemptionTypeList	PreemptionTypeList.FULLY_PREEMPTABLE	The preemptability of a task
type	string	-	The type of a task
token_name	list(str)	list("__<name>__"]	The list of names of the token which would be fired after completion of the task instance.

Table 2: *Keywords* to instantiate a task object.

3 Task Initialization for t1

In order to create a task instance, a programmer first needs to specify the granularity of the task. A task can be speficified as either *terminal* or *composite*. This information comes from the feature diagram. Secondly, she has to define each mandatory keyword that is inevitable to create task instance. The task model consists of many sub-feature models. It is necessary to expalin these branches to make a programmar familiar with the terminology.

- Granularity ($\mathcal{G}$): The granularity of a task (*terminal* or *composite*).
- Timing (T): This attribute includes all the relevant time-related task properties such as *release time*, *execution time*, *deadline*, and *period* information.
- Requirement ($\mathcal{R}$): The requirements of tasks that are *Resource Requirement* and *Deadline Requirement.*
- Priority (ρ): The priority information of a task.
- Dependency (Δ): The dependency relation with respect to other tasks, which is optional.
- Preemptable ($\mathcal{P}_\tau$): The preemptability property of a task, which is optional.
- Objective ($\mathcal{O}_\tau$): This attribute holds the task-related objective information, which is optional.

In the second phase, a programmer should specify the values for each attribute in the feature model. The keywords and correlated information is represented in Table 2

4 Task Initialization for t2