mad Documentation

helene coullon

Sep 20, 2018

Contents:

This documentation gives a complete guide to install and use MAD, the Madeus Application Deployer. MAD is a PYTHON implementation of the Madeus model. Its purpose is to offer a way to define a low-level deployment process and coordinate this deployment in an efficient and reliable manner. MAD is low-level but offers a highly generic deployment model.

CHAPTER 1

Installation

1.1 Code

Madeus is available online on `https://gitlab.inria.fr`. Anyone can create an account on this platform and download the source code for free. Please follow this direct link to the repository.

1.2 Installation

- MAD needs `python3` to run.
- It is more convenient to set the python environment variable `export PYTHONPATH=/path/to/mad:$PYTHONPATH`. You can also put this command inside your `.bashrc` to get MAD permanently available.

1.3 Check installation

- Go to your MAD directory
- Move to the examples directory `cd examples/user_providers`
- run this command: `python3 deploy_user_provider.py`

If everything went well, you should observe the following output:

```
[Mad] Assembly checked
[Mad] Start assembly deployment
[provider] Start transition 'init' ...
[provider] End transition 'init'
[provider] In place 'initiated'
[provider] Start transition 'config' ...
[provider] End transition 'config'
```

(continues on next page)

(continued from previous page)

```
[provider] In place 'configured'
[Assembly] Enable connection (user, ipprov, provider, ip)
[user] Start transition 'init' ...
[provider] Start transition 'start' ...
[provider] End transition 'start'
[provider] In place 'started'
[Assembly] Enable connection (user, service, provider, service)
[user] End transition 'init'
[user] In place 'initiated'
[user] Start transition 'config' ...
[user] End transition 'config'
[user] In place 'configured'
[user] Start transition 'start' ...
[user] End transition 'start'
[user] In place 'started'
[Mad] Successful deployment
```

This example will be explained in detail in the Getting Started.

CHAPTER 2

Getting Started

In this section we will study the example `examples/user_providers/deploy_user_provider.py`.

In MAD a deployment process is described under the form of an *assembly* of *components*. A component represents a software part to deploy. An assembly of components represents how components are connected through their dependencies.

The studied examples is composed of three different files:

- `provider.py` the provider component
- `user.py` the user component
- `deploy_user_provider` the assembly of component de deploy and its automatic deployment

First, the description of a component deployment will be explained. Second, the description of an assembly will be detailed.

2.1 Component

The deployment of a component is described as a kind of petri net structure. The deployement of a component is composed of:

- a set of **places** representing the different states of the deployment of the component,
- a set of **transitions** representing the different actions to perform to move from one place to another,
- a set of **dependencies** representing the different dependencies provided or required by external components and bound to internal places or transitions.

Lets focus on the component description `provider.py`:

```
from mad import *
from transitions import *
```

(continues on next page)

(continued from previous page)

```
class Provider(Component):

  def create(self):
    places = [
      'waiting',
      'initiated',
      'configured',
      'started'
    ]

    transitions = {
      'init': ('waiting', 'initiated', self.init),
      'config': ('initiated', 'configured', self.config),
      'start': ('configured', 'started', self.start)
    }

    dependencies = {
      'ip': (DepType.DATA_PROVIDE, ['configured']),
      'service': (DepType.PROVIDE, ['started'])
    }

  def init(self):
      time.sleep(10)

  def config(self):
      time.sleep(5)

  def start(self):
      time.sleep(5)
```

One can note in this example that a component is a new `class` that inherits the already existing `Component` class of MAD. For this reason, it is needed to import `mad` objects.

It is asked to the MAD developer to instanciate the abstract method `create` inside her new Component class. In this method will be initiated three data structures:

- `places` which is a simple list of strings. Each string representing the name of a given place.
- `transitions` which is a dictionary representing the set of transitions.
- `dependencies` which is a dictionary representing the set of dependencies of the component and their bindings to internal places and transitions.

2.1.1 Places

The `Provider` component contains four places, namely 'waiting', 'initiated', 'configured' and 'started'.

2.1.2 Transitions

The `Provider` component contains three transitions defined within a dictionary. The key of the dictionary is a string representing the name of the transition. Thus, `Provider` transitions are 'init', 'config', and 'start'. One can note that in this example transitions names are verbs. These names have been chosen to represent the fact that a transition is an action to perform.

Each key in the dictionary is associated to a triplet `(source, destination, action, [arguments])` where:

- *source* is the name of the source place of the transition
- *destination* is the name of the destination place of the transition
- *action* is a functor
- *arguments* is an optional tuple containing the arguments to give to the functor

For example, the `'init'` transition of `Provider` action is the internal method `self.init`, the source place is `'waiting'`, and the destination place is `'initiated'`. In other words, `'init'` is applied when moving from `'waiting'` to `'initiated'`, by performing the action associated to the method `init`.

The `init` method is defined within the component.

```
def init(self):
    time.sleep(10)
```

Inside the transition method is implemented the action to perform within the transition. In our example (see `transitions.py`) all functions contains a `sleep` system call to emulate that the action takes sometime to run.

Note: Transitions functors can be defined outside the component definition as it will be shown in other examples. However, the component object contained in *self* will be usefull for many reasons, as in `deploy_up_read_write.py` for instance. For this reason, it is more convenient to define transitions methods inside the component.

2.1.3 Dependencies

Dependencies of a component are defined as a dictionary. Each key of dictionary elements represents the name of a given dependency. For example, `Provider` component has two dependencies `'ip'` and `'service'`.

Each key is associated to a pair `(type, list_bindings)` where:

- *type* is the type of dependency
- *list_bindings* is a list of transitions or places to which the dependency is bound

Four types of dependencies are available in MAD:

- `DepType.USE` represents a *use* dependency, meaning that the component needs to use an external service provided by another component during its deployment process.
- `DepType.PROVIDE` represents a *provide* dependency, meaning that the component provides to external components some services during its deployment process.
- `DepType.DATA_USE` represents a *data-use* dependency, meaning that the component needs to use an external data provided by another component during its deployment process.
- `DepType.DATA_PROVIDE` represents a *data-provide* dependency, meaning that the component provides to external components a data during its deployment process.

The only difference between a service and a data is that once delivered a data is always available while a service could be disabled.

Each dependency is bound to a list of places or transitions. `DepType.USE` and `DepType.DATA_USE` can be bound to transitions only. Actually, an action of a transition may need a service or a data provided by external components. On the opposite, `DepType.PROVIDE` and `DepType.DATA_PROVIDE` can be bound to places only. When reaching a place a component is able to provide a data or a service to external components.

For example, `Provider` contains two dependencies:

- `'ip'` is a `DepType.DATA_PROVIDE` dependency bound (*i.e.* used) to the transition `'configured'`
- `'service'` is a `DepType.PROVIDE` dependency bound (*i.e.* used) to the transition `'started'`

Note: One can note that more than one place can be bound to a `DepType.PROVIDE` or `DepType.DATA_PROVIDE` dependency. When more than one place is given, a group is created and will be illustrated in advanced examples of this documentation.

Attention: MAD is a low-level deployement tool. It is asked to the developer to precise dependencies between the different components, however, the developer has the responsability to handle real communications between components. The developer has the liberty to choose the best way to do it, through environment variables and ssh connections, through file transfers, through RPC calls etc. Many libraries are available in Python3. This will be illustrated in advanced examples.

2.1.4 Graphical representation

Here is a graphical representation of the `Provider` component.

Note: If you take a look at the formal Madeus model, you will notice differences with the definition of a MAD component. Indeed, MAD simplifies a bit the component definition by omiting the *dock* concept which is automatically inferred and handled by MAD.

2.1.5 User component

In the *user-provide* example another component is declared: `user.py`.

```
from mad import *
from transitions import *


class User(Component):

  def create(self):
      self.places = [
          'waiting',
          'initiated',
          'configured',
          'started'
      ]

      self.transitions = {
          'init': ('waiting', 'initiated', self.init),
          'config': ('initiated', 'configured', self.config),
          'start': ('configured', 'started', self.start)
      }

      self.dependencies = {
          'ipprov': (DepType.DATA_USE, ['init']),
          'service': (DepType.USE, ['config', 'start'])
      }
```

(continues on next page)

(continued from previous page)

```
  def init(self):
      time.sleep(10)

  def config(self):
      time.sleep(5)

  def start(self):
      time.sleep(5)
```

With previous explanation you should be able to understand this component definition. The main difference with the `Provider` component is the type of dependencies: `DepType.DATA_USE` and `DepType.USE`. These dependencies are bound to transitions instead of places as they are used during deployment actions. The `User` component will be connected to the `Provider` component in the assembly. This will be detailed below.

Note: As for *provide* dependencies with multiple places, more than one transition can be bound to `DepType.DATA_USE` and `DepType.USE`. In this case more than one transition use the service or the data provided by external components.

2.2 Assembly of components

The file `/path/to/MAD/examples/user_providers/deploy_user_provider.py` contains the assembly of components as well as its run.

```
from mad import *

from provider import Provider
from user import User

if __name__ == '__main__':

  # Composant User
  user = User()

  # Composant Provider
  provider = Provider()

  ass = Assembly()
  ass.addComponent('user', user)
  ass.addComponent('provider', provider)
  ass.addConnection(user, 'ipprov', provider, 'ip')
  ass.addConnection(user, 'service', provider, 'service')

  mad = Mad(ass)
  mad.run()
```

2.2.1 Assembly description

In this example, the assembly is directly declared into the main function. This choice, of course, is left to the developer.

First, it is needed to import both mad and the components previously declared.

An assembly of components is composed of:

- instanciations of Component objects
- connections between components instances

Note: As an components and assemblies are defined in Python, the developer is free to pass additional arguments for object creations. This will be illustrated in advanced examples.

A Component, as previously detailed is a class. Its instanciation is a class instanciation.

```
user = User()
```

An assembly is also a class instanciation. The `Assembly` class is available in MAD.

```
ass = Assembly()
```

Instanciated components then need to be added to the assembly by calling `addComponent`. This method takes a string representing the name of the component, and the component object to add.

```
ass.addComponent('user', user)
```

Finally connections need to be added between components by calling the method `addConnection` of the assembly object.

```
ass.addConnection(user, 'ipprov', provider, 'ip')
```

A connection is composed of:

- a first component to connect
- the dependency name of the first component that will be connected
- a second component to connect
- the dependency name of the second component that will be connected

At this stage, it is important to understand that:

- `DepType.DATA_USE` dependencies can only be connected to `DepType.DATA_PROVIDE` dependencies
- `DepType.USE` dependencies can only be connected to `DepType.PROVIDE` dependencies

Note: If a bad connection is made MAD will print an error at runtime like this:

```
ERROR - you try to connect dependencies with incompatible types. DepType.USE
and DepType.DATA-USE should be respectively connected to DepType.PROVIDE and
DepType.DATA-PROVIDE dependencies.
```

2.2.2 Graphical representation

The graphical representation of the assembly of component is as follows:

2.2.3 Run the assembly

Once the assembly is created it can be run. To do so a `Mad` object is instanciated taking as argument the assembly object. The `run` methode of the `Mad` object is then called.

```
mad = Mad(ass)
mad.run()
```

To run this example you need to do

```
cd ``/path/to/MAD/examples/user_providers/``
python3 deploy_user_provider.py
```

The following output will be printed

```
[Mad] Assembly checked
[Mad] Start assembly deployment
[provider] Start transition 'init' ...
[provider] End transition 'init'
[provider] In place 'initiated'
[provider] Start transition 'config' ...
[provider] End transition 'config'
[provider] In place 'configured'
[Assembly] Enable connection (user, ipprov, provider, ip)
[user] Start transition 'init' ...
[provider] Start transition 'start' ...
[provider] End transition 'start'
[provider] In place 'started'
[Assembly] Enable connection (user, service, provider, service)
[user] End transition 'init'
[user] In place 'initiated'
[user] Start transition 'config' ...
[user] End transition 'config'
[user] In place 'configured'
[user] Start transition 'start' ...
[user] End transition 'start'
[user] In place 'started'
[Mad] Successful deployment
```

This output illustrates that the coordination of the deployment process is well handled by MAD. Let's take a closer look.

First, one can note that the `provider` component instance is the first one to start its execution. This is due to the fact that the `user` component must wait its data dependency to start its first transition `'init'`.

The data dependency of `user` is connected to the data provided in the place `'configured'` of `provider`. Thus, the connection is enabled by MAD when `provider` reaches its `'configured'` place. Then the `user` component can start its first transition.

When the component `provider` reaches its place `'started'` MAD enable the service connection.

All this coordination process is handled by MAD as well as the parallelism between transitions. This coordination is possible because of the dependencies declared by the developer.

CHAPTER 3

Advanced examples

CHAPTER 4

Developers documentation

4.1 MAD Engine

class engine.**Mad**(*ass*)

This class is the engine of the Madeus model. When running it first check if the assembly has warnings, then it calls the mad_engine method to run the semantics on the assembly.

check_warnings()

This method is called before runing an assembly. It checks components and their connections. The default behavior is to block the run if some warnings are detected.

Returns True if no WARNINGS, False otherwise

mad_engine(*dryrun*, *printing*)

This is the main function to run the operational semantics of the Madeus formal model. This is the heart of the coordination engine.

run(*force=False*, *dryrun=False*, *printing=True*)

This method run the assembly after checking its validity

4.2 Assembly

class assembly.**Assembly**

This Assembly class is used to create a assembly.

An assembly is a set of component instances and the connection of their dependencies.

addComponent(*name*, *comp*)

This method adds a component instance to the assembly

Parameters **comp** – the component instance to add

addConnection(*comp1*, *name1*, *comp2*, *name2*)

This method adds a connection between two components dependencies.

Parameters

- **comp1** – The first component to connect
- **name1** – The dependency name of the first component to connect
- **comp2** – The second component to connect
- **name2** – The dependency name of the second component to connect

check_warnings()
This method check WARNINGS in the structure of an assembly.

Returns false if some WARNINGS have been detected, true otherwise

component_connections = {}
BUILD ASSEMBLY

disable_enable_connections(*printing*)
This method build the new list of enabled connections according to the current states of "activated" places (ie the ones getting a token).

Parameters **configuration** – the current configuration of the deployment

Returns the new list of activated connections

init_semantics()
This method activate the initial places of each component and builds the global self.act_places

is_finish()
This method checks if the deployment is finished

Parameters **configuration** – the current configuration of the deployment

Returns True if the deployment is finished, False otherwise

semantics(*dryrun*, *printing*)
This method runs one semantics iteration by first updating the list of enbled connections and then by running semantics of each component of the assembly. :param dryrun: boolean representing if the semantics is run in dryrun mode :param printing: boolean representing if the semantics must print output

class assembly.**Connection**(*comp1*, *dep1*, *comp2*, *dep2*)
This class is used by the assembly to store connections between components

4.3 Configuration

class configuration.**Configuration**(*pla*, *conn*)
This class represents a configuration of the Madeus formal model. However, unlike the formal model and for performance reasons, the global configuration only stores active places and connections. Actually, each component stores its local set of actives places, transitions, input docks and output docks. A configuration is used by the engine to know the state of a deployment.

get_connections()
This method returns the list of active connections of the configuration

Returns self.connections

get_places()
This method returns the list of active places of the configuration

Returns self.places

update_connections(*conn*)
This method updates the connections of the configuration

Parameters **conn** – new list of active connections

update_places(*pla*)
This method updates the places of the configuration

Parameters **pla** – new list of active places

4.4 Component

class component.**Component**
This Component class is used to create a component.

A component is a software module to deploy. It is composed of places, transitions between places, dependencies and bindings between dependencies and Places/transitions.

This is an abstract class that need to be override.

add_dependencies(*dep*)
This method add all dependencies declared in the user component class as a dictionary associating the name of a dependency to both a type and the name of the transition or the place to which it is bound.

- a 'use' or 'data-use' dependency can be bound to a transition
- a 'provide' or 'data-provide' dependency can be bound to a place

Parameters **dep** – dictionary of dependencies

add_dependency(*name*, *type*, *bindings*)
This method offers the possibility to add a single dependency to an already existing dictionary of dependencies.

Parameters

- **name** – the name of the dependency to add
- **type** – the type DepType of the dependency
- **binding** – the name of the binding of the dependency (place or transition)

add_place(*name*)
This method offers the possibility to add a single place to an already existing dictionary of places.

Parameters **name** – the name of the place to add

add_places(*places*)
This method add all places declared in the user component class as a dictionary associating the name of a place to its number of input and output docks.

Parameters **places** – dictionary of places

add_transition(*name*, *src*, *dst*, *func*, *args=()*)
This method offers the possibility to add a single transition to an already existing dictionary of transitions.

Parameters

- **name** – the name of the transition to add
- **src** – the name of the source place of the transition
- **dst** – the name of the destination place of the transition

- **func** – a functor created by the user
- **args** – optional tuple of arguments to give to the functor

add_transitions(*transitions*)
This method add all transitions declared in the user component class as a dictionary associating the name of a transition to a transition object created by the user too.

Parameters **transitions** – dictionary of transitions

check_connections()
This method check connections once the component has been instanciated and connected in an assembly. This method is called by the engine -> assembly

Returns True if all dependencies of a component are connected, False otherwise

check_warnings()
This method check WARNINGS in the structure of the component.

Returns False if some WARNINGS have been detected, True otherwise.

end_transition(*dryrun*)
This method try to join threads from currently running transitions. For joined transitions, the dst_docks (ie input docks of the assembly) are stored for the new configuration. Un-joined transitions are stored for the new configuration.

Parameters **dryrun** – to indicate if the assembly is executed in dryrun mode.

Returns return (still_running, new_idocks)

Elements of the returned tuple are the list of transitions still running and the list of new input docks resulting from finished transitions in a pair.

get_dependency(*name*)
This method returns the dependencies object associated to a given name

Parameters **name** – the name (string) of the dependency to get

Returns the dependency object associated to the name

get_places()
This method returns the dictionary of places of the component

Returns self.st_places the dictionary of places

getcolor()
This method returns the color associated to the current component

Returns the printing color of the component

getname()
This method returns the name of the component

Returns the name (string) of the component

idocks_in_place()
This method returns the list of new places enabled. These places come from their set of input docks, all ready.

Returns (new_place, still_idocks)

Elements of the returned tuple are the new list of new enabled places, and the list of input docks without modification

init_places()
This method initializes the initial activated places of the component in its local configuration self.act_places

init_trans_connections(*comp_connections*)
This method initializes the dictionary associating one transition to a set of connections. This dictionary is used to start a transition. :param comp_connections: the list of all connections associated to the current component.

isconnected(*name*)
This method is used to know if a given dependency is connected or not :param name: name of the dependency :return: True if connected, False otherwise

place_in_odocks(*my_connections*)
This method represents the one moving the token of a place to its output docks.

Parameters **my_connections** – the list of enabled connections of the current

component. :return: (new_odocks, still_place)

Elements of the returned tuple are the new list of output docks and the list of places that cannot move to output docks (possible because of groups and dependencies on services).

semantics(*my_connections*, *dryrun*, *printing*)
This method apply the operational semantics at the component level. It takes as input the current configuration of the deployment which represents runtime information.

Parameters **configuration** – The current configuration of the deployment

Returns a tuple (new_transitions, new_places, new_idocks, new_odocks)

Elements of the returned tuple are respectively the list of components, transitions, places, input docks and output docks in the new configuration of the current component.

setcolor(*c*)
This method set a printing color to the current component

Parameters **c** – the color to set

setname(*name*)
This method sets the name of the current component

Parameters **name** – the name (string) of the component

start_transition(*my_connections*, *dryrun*)
This method start the transitions ready to run:

- source dock of the transition in the list of activated output docks
- all dependencies required by the transition in an activated connection

Parameters

- **my_connections** – list of connections associated to the current component
- **dryrun** – to indicate if the assembly is executed in dryrun mode.

Returns (new_transitions, still_odocks)

Elements of the returned tuple are the list of new transitions started by the method and the list of output docks still waiting for connections.

class component.**Group**
This class is used to create a group object within a Component. A group is a set of places and transitions to which a service provide dependency is bound. This object facilitate the semantics and its efficiency.

4.5 Place

class place.**Dock**(*place*, *type*, *transition*)
This Dock class is used to create a dock.

A dock is an input or an output of a place.

getplace()
This method returns the Place object associated to the dock

Returns a Place object

gettype()
This method returns the type of dock

Returns self.DOCK_TYPE

class place.**Place**(*name*)
This Place class is used to create a place of a component.

A place represents an evolution state in the deployment of a component.

add_provide(*dep*)
This method is used to add a provide dependency bound to the current place

Parameters **dep** – the provide dependency to add

create_input_dock(*transition*)
This method creates an additional input dock to the current place. This method is called by Transition. An input dock of a place corresponds to a destination dock of a transition.

Parameters **transition** – the transition to which the dock will be associated

Returns the new input dock

create_output_dock(*transition*)
This method creates an additional output dock to the current place. This method is called by Transition. An output dock of a place corresponds to a source dock of a transition.

Parameters **transition** – the transition to which the dock will be associated

Returns the new output dock

get_inputdocks()
This method returns the list of input docks of the place

Returns self.input_docks

get_outputdocks()
This method returns the list of output docks of the place

Returns self.output_docks

get_provides()
This method returns the list of provide dependencies bound to the current place

Returns self.provides

getname()
This method returns the name of the place

Returns name

4.6 Transition

class transition.**Transition**(*name*, *src*, *dst*, *func*, *args*, *places*)
This Transition class is used to create a transition.

A transition is an action, represented by the 'run' function. This action is performed between a source and destination dock, each of which is attached to a place.

This is an abstract class that need to be override. In particular, the method 'run' must be override

bind_docks(*places*)
This method is called from the Component class to create docks into places associated to the transition. Once created these docks are bound to the transition.

Parameters **places** – the dictionary of all places declared for the component

get_dst_dock()
This method returns the destination dock of the transition

Returns the destination dock self.dst_dock

get_src_dock()
This method returns the source dock of the transition

Returns the source dock self.src_dock

getname()
This method returns the name of the transition

Returns name (string)

join_thread(*dryrun*)
This method tries to join self.thread. The default behavior has no timeout.

Returns True if the tread has been joined, False othwise

setname(*name*)
This method set the name of the transition

Parameters **name** – tha name (string) to set

start_thread(*dryrun*)
This method creates the thread of the transition

Returns the Thread of the transition

4.7 Dependency

class dependency.**DepMandatory**
This class is not instanciated and is an Enumeration. It is used to know if a dependency is mandatory or optional.

class dependency.**DepType**
This class is not instanciated. It handles the types of dependencies.

class dependency.**Dependency**(*name*, *type*, *bindings*)
This class represents a dependency.

connect()
This method set self.free to False to indicate that the dependency has been connected in the assembly. Note that a dependency can be connected more than once, however this method is used to throw a warning when dependencies are not connected.

Returns self.free

connectwb(*wb*)
This method set self.free to False to indicate that the dependency has been connected in the assembly. Note that a dependency can be connected more than once, however this method is used to throw a warning when dependencies are not connected.

Returns self.free

getbindings()
This method returns the place or the transition to which the dependency is bound. If the dependency is of type DepType.USE or DepType.DATA_USE it is bound to a transition, otherwise it is bound to a place.

Returns the transition or the place self.binding

getname()
This method returns the name of the dependency

Returns name

gettype()
This method returns the type of the dependency DepType

Returns type

isfree()
This method indicates if the dependency is free or not, ie if it is already connected to another dependency within the assembly

Returns True if not connected, False if free

4.8 Utility

class utility.**Messages**
This class is not instanciated. It is used for valid, warning, and fail color-printed messages.

CHAPTER 5

License

MAD is distributed under a GNU GENERAL PUBLIC LICENSE Version 3, 29 June 2007.

CHAPTER 6

Publications

Madeus: A formal deployment model. Maverick Chardet, Hélène Coullon, Christian Perez, Dimitri Pertin. *International Symposium on Formal Approaches to Parallel and Distributed Systems* (4PAD 2018) hosted at HPCS 2018. [Download] [Cite]

Python Module Index

Index

J

M

P

R

S

T

U