Simulator Configuration Language

The SimCfg Configuration

This documents gives a brief overview over the SimCfg configuration language. SimCfg is now fully functional and implemented in the PeerfactSim.KOM simulator. It uses a domain specific language for configuration and provides a richer feature set, compared to the XML-based configuration. The syntax and the use of different statements is described in the section "Sections".

A working example for a configuration file can be found at config/mobile-aggr/blocktree-750m.simcfg or /config/mobile/squatter/squatter.simcfg

Features

SimCfg brings multiple new features to the simulator configuration. Those include the following:

- Slimmer configuration syntax
- Modular config files through the use of a more expressive import statement with arbitrary import depth
- The use of flags for enabling or disabling components
- Support for variations to modify settings and flags with the use of one simple parameter
- Component configuration with @Configure annotation
- Dependencies with the @After annotation
- Dynamic attributes
- Eclipse plugin
  - Syntax highlighting
  - Auto completion for import statements, component names, and variables
  - Error highlighting for missing or wrong mappings

Component configuration

Like the DefaultConfigurator the SimCfgConfigurator can call a specific method after all components have been created and all setters have been called. This allows components that depend on each other to perform component specific configurations. Using the DefaultConfigurator this is done by providing a method with the signature 'public void _verifyConfig'. SimCfg allows the use of an arbitrary method for this purpose. Provided it adheres to the following criteria:

- Annotated with @Configure

- Return type of void or boolean
- Either no parameters or a combination of the following or their subtypes:
  - HostBuilder
  - ParsedConfiguration
  - Configurator

Should the methods return type be void, it will only be called once. A configuration method with a return type of boolean will be called every round until it returns true. A round is the call of a configure method on every component that has not yet been successfully configured.

It might often happen that one component depends on another component for its operation, or supports the use of input from another component. Thus SimCfg provides the @After annotation to be used in conjunction with the @Configure annotation. The @After annotation supports the use of optional and required dependencies. An optional dependency is only minded if a component of the type is actually configured. Required dependencies must be present, otherwise an exception will be thrown.

The configurator will automatically add an optional dependency to the HostBuilder for every ComponentFactory with an @Configure annotation. This ensures that all overlay nodes have received all their configured components before the component specific configuration is executed.

Examples

```
// The configuration method is called once after a class of the
// type ObstacleModel has been configured if present, otherwise
// it is called based on its position in the configuration file.
@Configure
@After(optional={ObstacleModel.class})
public void configure(Configurator configurator) {

}

// The configuration method is called once after a class of the
// type WaypointModel has been configured. An exception
// is thrown if no such class is configured in the SimCfg file.
@Configure
@After(required={WaypointModel.class})
public void _verifyConfig() {

}

// The combination of optional and required is supported as well.
// Furthermore, multiple dependencies can be specified using a
// comma as delimiter.
```

Dynamic Attributes

Components can now have limited support for dynamic attributes. Meaning that they can be configured without the use of specific setters for each attribute. In case that SimCfg cannot find a setter with the name required for a configured attribute, it will first check for the existence of a method annotated with @UnknownSetting. The required signature is ‘public void someName(String, Value)’ or ‘public void someName(String, String)’. In the second case the value is converted to a String. This might lead to a loss of precision or might just be inconvenient. Thus SimCfg can pass the raw Value provided by the Xtext parser. The component can then identify the required type itself and use the convertTo method of the SimCfgTypeConverter to convert the Value to the required type. In such a case the automatic resolving of in-string variables is currently not supported.

Supported types of the SimCfgTypeConverter are: boolean, byte, Class, Class[], double, int, long, short, String, String[]. The conversion can be done independently of the actual type, but might yield better results if the type is known.

Sections

In this sections the syntax and features of the SimCfg configuration file sections are described. SimCfg requires the sections to be exactly in the order as given in this document. Sections that not confirm to the order will be ommited or break the configuration file.

Note on the syntax:

The syntax requires keywords to be escaped if used in another context. Configuring a component with a setter named setSettings will require the use of the following code block:

```
settings {
    some_attribute: 5
}
```

As settings is a keyword this will lead to a parsing error. Thus it has to be escaped by the use of ^. The working configuration block thus would look like the following:

```
^settings {
    some_attribute: 5
}
```

The preferred way of defining settings and naming setters is the underscore notation. Even though this is not necessary, it looks way better :-) In case of setters SimCfg will remove the underscores and convert the identifier to a camel case notation internally. Settings on the other hand are not converted and are case sensitive.

Imports

SimCfg allows the import of other SimCfg configuration files. The imports are similar to the XML tag, but allow a selective import of the other configuration file. The import statement allows two modes for specifying the configuration file:

- **File: '..\config\other.simcfg'**

  In this mode the config file will simply be loaded from the given location. The path is expected to be relative to the current configuration file.

- **Qualified name: 'default.linklayer'**

  The qualified name will be translated to a directory path below the config/ directory. Using the qualified name from the example this would be 'config\default\linklayer.simcfg'.

The import statement allows different parts of the configuration file to be imported. See the following list for the available options:

- everything
- noconfig (everything except the configuration)
- imports
- mappings
- flags
- settings
- variations
- defaults
- configuration

Examples These are some examples of possible imports:

```
import everything from file '..\\config\\mobile.simcfg'
import \^defaults from file '..\\config\\mobile.simcfg'
import mappings from qf default.mappings.linklayer
import everything from qf default.maps
import \^defaults from qf default.maps
import \^imports, \^flags from qf default.maps
```

Mappings

Mappings are used to assign simple names to specific classes. This is to avoid the use of the class name on every defined component. Using mappings is mandatory in SimCfg configuration files.

A mapping consists of the keyword *map*, followed by an alias, the keyword *to*, and the fully qualified name of the mapped class.

Examples These are some examples for valid mappings:

- map simulator_core to de.tud.kom.p2psim.impl.simengine.Simulator
- map topology to de.tud.kom.p2psim.impl.topology.TopologyFactory

Flags

Flags can be used to enable or disable single or multiple components in the configuration. Every line in the flags section is one flag. Using ! in front of a flag can be used to disable flags that were enabled in imported configurations. By default all flags are disabled, but configurations will provide default flag configurations.

Examples

```
flags {
    churn
    enerygMeasurement
    !moveSlaw
    moveGauss
}
```

Settings

The settings section is used to define variables that can later be referenced in the configuration section by either using $variableName or ${variableName} inside a string.

Variables (as well as component attributes) can be of different types:

- **String:** ‘some text’
- **Integer:** 42
- **Double:** 42.0
- **Time (long):** 2h / 2m / 2ms
- **Boolean:** ‘true’ / ‘false’ or 1 / 0
- **StringList:** [‘some’, ‘text’]

Supported types in constructors and setters are: String, String[], int, double, long, boolean, short, byte, Class, Class[]. The SimCfgTypeConverter will print a warning to the log if one of the conversions may lead to faulty results or a loss of precision.

Variables of the type String support the use of inline variables, as mentioned before. This feature will try to resolve variables in Strings until all variables have been resolved. Use this with care, as multiple levels of inline variables have not been tested yet.

Examples

- measurement_start: 90m
- churn: ‘true’
- lambda: 5.0
- component_classes: [‘de.tud.kom.p2psim.impl.service.aggr.gossip.manet.MANETOverlayNode’]
- component_class: ‘de.tud.kom.p2psim.impl.service.aggr.gossip.manet.MANETOverlayNode’
- text: ‘Churn is set to ${churn}’

Defaults

The defaults section can be used to assign default attributes to specific component types. Attributes that are assigned to a component type in the defaults section will automatically be applied to all configured components of that type unless they override that attribute.

Note that variables in imported defaults will be resolved against the aggregated list of settings from all imported configuration files. Thus make sure the used variable names are either unique or don’t use variables in the defaults section to avoid unexpected behavior. Although this behavior may be wanted, see the $phy assignment in the example. It can be overriden by either overriding the setting in a defaults section of a higher level configuration file or by redefining the variable $phy in the settings section.

Important: Defaults of the same component will be merged with defaults of imported components.

Examples

```
defaults {
    simple_mac {
        phy: $phy
        max_retransmissions: 3
        traffic_queue_size: 100
        traffic_queue_timeout: 1s
    }
}
```

Variables

SimCfg supports multiple variation sections that can be used to modify loaded configurations. Variations can turn flags on or off and override settings. Note that the variation name may not contain spaces and that the flags and settings block are required to be in the same order as in the example.

Examples

```
variations {
    GaussMovement {
        flags {
            !moveSlaw
            moveGauss
            obstacles
        }
        settings {
            world_x: 750
            world_y: 750
        }
    }
}
```

Configuration

The configuration section is used to define the components that should be initialized by the configurator. The SimCfgConfigurator knows four types of components:

- **Static Component**

  A static component is basically a singleton and gets created by invoking the method with the signature 'public T getInstance()'. Unlike in the XML configuration this method is hard coded and cannot be changed. Unlike other components the definition of a static component will not override definitions in imported configuration files. This allows to add multiple analyzers to the same monitor throughout multiple configuration files. The SimCfgConfigurator will print a warning in such a case.

- **Factory**

  Factories are used to create different layers for the hosts. Every factory must implement the ComponentFactory interface whos createComponent method will be called for every host.

- **Helper Component**

  The helper components are just plain classes that are initialized. They should implement the Composable interface or provide a method annotated with @Configure. Otherwise the class will not be called and would just be hold in memory without purpose.

- **Component**

  Every component that is not on the top level of the configuration falls into this category. Like helper components they are just pojos but are assigned to their parent component via a setter composed of 'set' and their name.

The use of flags Examples using a factory with flags test1 and test2:

- **factory** some_factory **when** test1 { }
- **factory** some_factory **when** !test1 { }
- **factory** some_factory **when** test1 && !test2 { }
- **factory** some_factory **when** test1 && test2 { }
- **factory** some_factory **when** !(test1 || test2) { }

The flags in the when clause of the components are boolean expressions. Internally they are evaluated using the JavaScript engine provided by the Oracle JRE.

Multiple factories Since factories can be defined multiple times the naming will be ambiguous. To avoid such a problem and to ensure that the right factory is referenced in a group of hosts, the factories can be named. To name a factory an alias has to be inserted after the factories mapping name and before the use of any flags. Should the alias be omitted, the mapping name will be used to identify the factory.

Examples

```
factory some_factory as overlay when test1 { }
```

Other components are declared by the same rules as factories, except that they lack an alias part and use different keywords for identification.

Examples

```
static oracle { }
helper host_builder { }
```

Simple components also use this schema, but use the setter name instead of a component identifier like static, factory, or helper.

Examples

```
analyzer bandwidth_analyzer { }
churn_model exponential_churn { }
```

IDE intergration

At the moment SimCfg can be used in Eclipse and IntelliJ IDEA. In the latter case only syntax highlighting is supported using the provided IDE functionality.

Eclipse

The support for Eclipse is based, as the configurator itself, on Xtext. As such it requires the Xtext plugin to be installed in the IDE. It can be found in the Eclipse marketplace. The plugins for the SimCfg language are in the PeerfactSim.KOM repository. These have to be installed by placing them in the Eclipse dropins directory. The dropins directory can be found in the Eclipse directory on Windows and at /usr/share/eclipse/dropins on Linux.

Xtext: http://marketplace.eclipse.org/content/xtext

Plugins: PeerfactSim.KOM-SimCfg/trunk/plugins

Note: Eclipse might ask if you want to add the Xtext Nature to the project, confirm this with yes to enable support for Xtext based plugins in the project.

Limitations At the moment the plugin still contains some minor bugs, e.g., a not fully working autocompletion when using the file based import and additional keyword entries with the qf based import.

IntelliJ IDEA

There is currently no plugin for IntelliJ IDEA, but fortunately the IDE supports syntax highlighting for custom languages. To enable SimCfg syntax highlighting go to File -> Settings... -> File Types. There click on the green plus sign and use the following settings:

- **Name:** SimCfg
- **Description:** Simulator Configuration Language
- **Line comment:** //
- **Block comment start:** /*
- **Block comment end:** */
- **Support paried braces:** Yes
- **Support paried brackets:** Yes
- **Support paried parens:** Yes
- **Support string escapes:** Yes
- **Keywords:** import, qf, file, from, map, to, flags, settings, defaults, variations, configuration, static, factory, helper

Afterwards select the entry Simulator Configuratino Language and add the file extension *.simcfg to the registered patterns.

Netbeans

Netbeans seems to require an extra plugin like eclipse for syntax highlighting and thus is not supported at the moment.