

# Cushion for SOFA 2

## Table of contents

1 Overview.....	2
2 Cushion commands.....	2

## 1. Overview

**Cushion** is a command line tool for developing SOFA 2 components. It is delivered as a separated archive (see [Download](#)). It contains directory `bin`, in which the launching script reside (there are two variants of the script: `cushion.sh` for the UNIX-like systems and `cushion.bat` for Windows systems).

The first parameter of the Cushion represents a command which has to be executed.

The Cushion works with a current directory; it stores in it all generated files and also its own configuration, which is located in the `_cushion` file (do not edit the file by hand). A typical usage scenario is to develop a single application (i.e. set of components forming a single application) in a single directory. This directory can be seen as a current working project.

## 2. Cushion commands

- **help** prints a short help and all available commands.  
Use `cushion help <command>` for help on a specific command.
- **new** creates new interface, frame, or architecture.

General usage is:

```
cushion new <interface, frame, architecture> <initial, next, branch>
name [previous_version]
```

- 2nd parametr specifies what has to be created.
- 3rd parametr defines, which version has to be created. `initial` means to create an initial version of the given name. It will be successfull only if there is no version of the given name. `next` means to create a subsequent version of the version given as a 5th parameter. `branch` means to create a subsequent version in a new branch.
- 4th parametr is the name of the element to be created.
- 5th parametr has to be specified only if a subsequent version (`next` or `branch`) has to be created.

The `new` command creates a new element in the repository and also generates a new ADL file in a subdirectory of the current directory. The subdirectory has the same name as the created element; the ADL file is named as `adl.xml`.

This ADL file should be completed and changes subsequently committed to the repository by `commit` command.

- **commit** command commits changes of the working elements to the repository.  
General usage is:

```
cushion commit [name_of_the_committed_element]
```

Without 2nd parameter, it commits changes of all working elements.

Note: it commits just changes in ADL files.

- **checkout** command checks out an entity from the repository for further editing. General usage is:

```
cushion checkout name [version]
```

Without the version identifier, it "guesses" a version.

- **update** command updates working elements from the repository. General usage is:

```
cushion update [name]
```

Without the name it updates all working elements. **WARNING** - after executing this command, all uncommitted changes will be lost!

At first glance, this command is similar to the `checkout` command but the `update` can be used only on elements previously created and/or checked out.

- **compile** command compiles Java code of a particular entity (interface type or architecture). The code has to be located in the `code` subdirectory of the particular entity directory. If the entity depends on other entities (e.g. architecture on interface types), these are checked out (if not yet checked) and their code is compiled as well (if it has not been compiled yet).

General usage is:

```
cushion compile [name]
```

Without the name it compiles all working elements (which can be compiled).

- **upload** command takes code of the elements (interface types or architectures) and uploads it as a codebundles to the repository.

General usage is:

```
cushion upload [name]
```

Without the name parameter, it uploads all working interface types and architectures. **NOTE:** it **DOES NOT** automatically (re)compile code before uploading.

- **assembly** creates new assembly descriptor. General usage is:

```
cushion assembly <initial, next, branch> name [previous_version]  
topLevelArchitecture [architecture_version]
```

- 2nd parametr has the same meaning as 3rd parameter of the new command
- the `topLevelArchitecture` is the name (optionally followed by the version) of the architecture which should serve as a top-level component in the application.

This command pregenerates the assembly descriptor. The user should continue and recursively assign the architectures of subcomponents. Do not forget after editing assembly descriptor to `commit` it.

- **print** command prints various information based on further parameters.
  - `allimplements` is currently the only supported parameter. It takes other two parameters, name of a frame and its version and prints all architectures that implement the frame. This action can be useful during filling in an assembly

descriptor.

- **deplplan** creates new deployment plan.

General usage is:

```
cushion deplplan <initial, next, branch> name [previous_version]
assembly_descr [assembly_descr_version]
```

Meaning of the parameters is the same as parameters of the `assembly` action; just this action takes an assembly descriptor from which creates the deployment plan. The user should then fill in the deployment plan (set names of deployment docks, where the particular components should be launched) and use the `deploy` action to generate connector according the deployment plan.

- **deploy** uses a deployment plan and generates connectors according to it. An application is ready to launch after execution of this action.

General usage is:

```
cushion deploy depl_plan_name
```

The deployment plan has to be already created (via the `deplplan` action) and filled in.

- **status** prints out all current working entities, their type and other information.
- **bcheck** verifies behavior compliance of architecture components.

General usage is:

```
cushion bcheck (dchecker|ebpspin) architecture
```

`dchecker` causes checking compliance using specification in original Behavior Protocols, while `ebpspin` enables verification of specification in Extended Behavior Protocols using the Spin model checker.

#### Note:

More information about behavior specification and verification can be found on [the formal methods documentation page](#) and on [the publication page](#).

- **jpfccheck** checks Java code of all primitive components in a given assembly (or a primitive architecture) with JPF.

General usage is:

```
cushion jpfccheck {assembly | primitive architecture} [properties file
with environment-related information]
```

This action is applied to the current version of a given assembly (or a primitive architecture) that is available in the working directory. Therefore, it is necessary to use the `checkout` action before `jpfccheck` to retrieve the assembly (or the architecture) from the repository.

The properties file can provide (i) the name of Java class with custom specification of method parameter values and (ii) the name of a directory where generated files should be stored. Default values are, respectively,

```
org.objectweb.dsrg.sofa.envgen.EmptyValueSet and <path to  
working directory>/_env.
```

Example of the properties file:

```
valuesets.class=org.objectweb.dsrg.sofa.envgen.EmptyValueSet  
temporary.directory=_env
```

Note also that this action does not work on components, whose Java implementation involves e.g. GUI or network communication, since JPF has limited support of Java classes that contain native methods. Currently, only the core classes like Thread and file I/O-related classes are supported.

- **exec** performs a list of cushion actions stored in a text file.

General usage is:

```
cushion exec <text file with a list of cushion actions>
```

The command executes a list of actions stored in the given text file. Each action is defined on a separate line and described only by its parameters (cushion keyword is omitted):

```
# this is a comment  
new interface initial foo.ILog  
  
new frame initial foo.FLog
```