SCULP Reference

1. Getting Started

The most básic program written in SCULP consists on a single procedure:

```
post("Hello world")
```

This program will post the message **Hello World** in the space where the program is posted. To see more about the different procedures that exist in SCULP, see the section 2. Procedures.

Additionally to the procedures, there are instructions used to build more complex programs:

```
when *."hello".* do post("Hey!")
```

This program will post the message **Hey!** the first time a message containing the word **hello** is found in the space where the program is posted. In this example, the conditional instruction *when* is used, this kind of instructions execute the given statement when the given constraint is satisfied. To see more about the constraint system build-in SCULP see the section 3. Constraints. To see more about the different instructions that exists in SCULP, see the section 4. Instructions

2. Procedures

The procedures are the final actions a program in SCULP execute.

2.1. post

Syntax: `post(message)`
Where `message` is a **string**. Posts the given `message` in the space where the program is executed.

2.2. signal

Syntax: `signal(message)`
Where `message` is a **string**. Posts the given `message` in the space where the program is executed, then it's deleted in the same time unit. Has result of this behavior, the message will not be present in the space but

2.3. notify

Syntax: `notify(message)`
Where `message` is a **string**. Posts the given `message` in the notifications space of the space where the program is executed. If the space correspond to the main space of a user, the `message` will appear in the notification's box of that user.

2.4. mail

Syntax: `mail(message)`
Where `message` is a **string**. Posts the given `message` in the mailbox space of the space where the program is executed. If the space correspond to the main space of a user, the `message` will be send to the email of that usar..

2.5. clock

Syntax: `clock(crontab)`
Where `crontab` is a **string**. Posts the given `crontab` in the clock space of the space where the program is executed. If the `crontab` syntax is correct, a cron that emits a **tick** signal in the clock space will be created. Any previously created cron in that space will be replaced. Using **0** as `crontab` will remove any cron existing for that space.

2.6. vote

Syntax: `vote(choice)`
Where `choice` is a **string**. Posts the given `choice` in the poll space of the space where the program is executed. This will only work if there is an open poll in that space. Users can vote once, following vote will be ignored.

2.7. create-poll

Syntax: `create-poll(question)`
Where `question` is a **string**. Starts a new poll in the space where program is executed if there isn't any open pull in that space. If there is a closed poll in that space, its votes will be removed.

2.8. close-poll

Syntax: `close-poll`
Closes a existing poll in the space where program is executed.

2.9. rm

Syntax: `rm(pid,user,body)`
Where `pid`, `user`, and `body` are **patterns**. Removes the posts that match the given `pid`, `user` and `body` patterns, unrequired fields can be filled with the match all pattern `*`. To see more about patterns, see the section 4.1. Patterns.

2.10. kill

Syntax: `kill(str)`
Where `str` is a **string**. Kills the process with the given `pid` if exists in the space where the program is executed. Use **all** as pid to kill all process in the space.

2.11. call

Syntax: `call(name)`
Where `name` is a **string**. Calls the process identified by `name` previously defined in the bin space of the current space (result of defining it with the def.. as instruction). It will send the signal `name` to the bin space of the current space. Any process waiting for that signal will be triggered.

3. Constraints

The constraint system used in DSpaceNet is based on pattern matching with a small extension called Message Matching operators that are used to do pattern matching to specific parts of a message.

3.1. Patterns

Given a and b two patterns:

Operator	Name	Example	Description
.	Pattern concatenation	a . b	Matches anything that a matches followed by anything that b match
*****	Match all	*	Match anything
?	Match once	?	Match any character once
v	Logical Or	a v b	Matches anything that a matches **OR** b matches

&	Logical And	a & b	Matches anything that a matches **AND** b matches

Additionally to the previous operators, string literals (delimited by “) can be used to match these specific strings, for example, the pattern `“hello”. *` will match any string that starts by the word hello followed any amount of characters. Keep in mind that patterns will always look to match the whole string, if a partial match is required (like in the previous example) you should prepend and/or append the operator `*`.

3.2. Message Matching Operators

Since user messages in DSpaceNet follow a specific structure, patterns alone are not enough to match specific parts of the message such as the message’s body or the user of the message.

Syntax: `{ field1:pattern1, field2:pattern2, ..., fieldN:patternN }`
Where `field1`, `field2` and `fieldN` are valid **field names**, `pattern1`, `pattern2` and `patternN` are **patterns**. Match any message which fields match their corresponding pattern. Currently available field names are listed below:

- **usr**: Match the user of the message
- **txt**: Match the body of the message
- **pid**: Match de PID of the message

Example: `{ usr:”frank”, txt:*.”?” }` will match any message from the user **frank** that ends in a **?** symbol.

4. Instructions

Instructions are used to alter the execution flow of the program, much like in other programming languages.

4.1. when … do ...

Syntax: **when** `constraint` **do** `statement`
Executes the `statement` if the `constraint` is satisfied, if not, the program will persist over time until the `constraint` will be satisfied.

4.2. whenever … do …

Syntax: **`whenever`** `constraint` **`do`** `statement`
Executes the `statement` every time the `constraint` is satisfied. The program will persist over time until it is killed.

4.3. while … do ...

Syntax: **`while`** `constraint` **`do`** `statement`
Executes the `statement` while the `constraint` is satisfied. The program will persist over time until the `constraint` no longer is satisfied.

4.4. if … then ...

Syntax: **`if`** `constraint` **`then`** `statement`
Executes the `statement` if the `constraint` is satisfied. If not, the program ends.

4.5. do … until …

Syntax: **`do`** `statement` **`until`** `constraint`
Executes the `statement` until the `constraint` is satisfied, then the program ends.

4.6. unless … next …

Syntax: **`unless`** `constraint` **`next`** `statement`
Executes the `statement` in the following time unit of when the `constraint` is satisfied. The program will persist over time until the `constraint` will be satisfied.

4.7. enter @ … do …

Syntax: **`enter`** **`@`** `space` **`do`** `statement`
Where `space` is a **string**. Enters to `space` inside the current space and executes the `statement`. If `space` is not a valid space name, the program ends.

4.8. exit @ … do …

Syntax: **`exit`** **`@`** `space` **`do`** `statement`
Where `space` is a **string**. Exits from `space` to the parent space of the current one and executes the `statement`. If `space` is not a valid space name or is not the name of the current space, the program ends.

4.9. repeat …

Syntax: `repeat statement`
Executes the `statement` in every time unit, until the program is killed.

4.10. next …

Syntax: `next statement`
Executes the `statement` in then next time unit.

4.11. def.. as …

Syntax: `def name as statement`
Where `name` is a **string**. Executes the `statement` every time the `name` signal is send to the bin space, for example using the call procedure.

4.12. Parallel Execution

Syntax: `statement || statement`
Executes both `statements` in the same time unit

4.13. Skip

Syntax: `skip`
Can be used in place of any statement to do a no operation, it is useful to skip the current time unit.