rbClips scaffold

Jarek 'Jarcec' Cecho

May 5, 2009

Contents

1 Introduction

This document describe scaffold structure of rbClips, binary extension of CLIPS for ruby scripting language. This is first scaffold so in many places, there is direct insertion of CLIPS code which will be hopefully removed in future versions.

General notes:

- Entire CLIPS environment lives inside it's own namespace (module) called 'Clips'.
- Most of objects have common methods with same semantic (if semantics differs, it's mention specifically)

  - `to_s()` Returns fragment of code that describe object in CLIPS word (it's syntactically correct CLIPS code, that can be useds)
  - `save()` Newly created instances/classes/facts/rules/... are not directly inserted into clips environment, but they have to be inserted manually throw this method. Be aware that almost in every case, saving entity to CLIPS means locking its object in ruby. Meaning that after saving, object is in read-only state and can't be altered.
  - `destroy()` Opposite function for save(), it will try to remove given entity from CLIPS environment.
  - `environment()` Return in which environment (instance of class Environmnet) is the object connected or nil if it's environment free (can be used anywhere).

- CLIPS build in types have lower case id that represent them in rbClips. To make reading easiest, for this document pseudovariable ClipsType is used and means on from :symbol, :string, :lexeme, :integer, :float, :number, :instance_name, :instance_address, :instance, :external_address, :fact_address.

1.1 Flow of rbClips application

In common CLIPS application normal flow of application is creating instances, rules and facst and then starting firing the rules. rbClips is in fact only wrapper above CLIPS, so it share the same approach. You defince classes, instances, facts, rules and save them to environment(s). Than simply run `Clips::Base.run(Fixnum)` to start firing rules.

2 Constraits

CLIPS provide strong mechanism for defining contraits (limitation) of slots in Templates and Classes. In rbClips it's surrounded by class Constrait. Saving Constrait in CLIPS environment is impossible because constraits can't live along in CLIPS, that's possible only in ruby environment (to create one constrait and than pass the same config to all slots). In most places, where the constrait object is requested is possible to pass Hash - it will be pass to the constructor and constrait object will be build in place.

Below are listen options for hash passed to constructor, note that only one hash key is valid!

- `:type => ClipsType | :any | [ClipsType, ...]` Specifing which type can be used
- `:values => Array` List with allowed values
- `:range => Range`
- `:cardinality => Range` Cardinality of multislot

3 Base

Function and action that are not exactly related to some bigger topic that is wrapped by another class listen below are accessible by object Base. It have just and only static methods for various use:

- `run(Fixnum, Environment | nil)` Run clips (start applying rules) in given Environment or when passed nil in actually set environment, this method will end after the maximal count of rules will be fired or there aren't any rule to rile.
- `reset` Reset clips environment to defaults
- `agenda` Return String containing outputs of agenda function
- `insert_code(String)` Insert given fragment of code to CLIPS directly (no checks, no abstraction)

- `static_constraint_checking` and `static_constraint_checking=(Boolean)` Get or set static constrait
- `dynamic_constraint_checking` and `dynamic_constraint_checking=(Boolean)` Get or set dynamic constrait
- More methods mentioned later in this document

3.1 Environment

rbClis is environment aware (can run multiple environments) - everythink operates with class Environment. Base class have methods to work with it:

- `getEnvironment()` Return current environment.
- `setEnvironment(Environment)` Set environment, this method returns previous environment instance

Please note that one environment (the default one) is created for you when loading rbClips into memory.

```
# Saving previous (automatically generated)
defenv = Clips::Base.getEnvironment
# Creating new environment
newenv = Clips::Environment.new
prevenv = Clips::Base.setEnvironment(newenv)
```

4 COOL

CLIPS object-oriented language description and its wrapping by ruby environment.

4.1 Classes

For interaction with classes in CLIPS, rbClips have class Class (yeah same name, just first letter is upper case). It have two static methods:

- `new()` Create new instance, description is below
- `load(String)` Load class from CLIPS environment and return it's representation in ruby.

For creating new classes, you need to create new instance of class 'Class':

```
animal = Clips::Class.new :name => 'animal'
```

Constructor have more keys in hash for params that follows defclass command:

- `:name => String` Name of class in CLIPS
- `:is_a => Array` Inheritance list, can contain:
  - `Clips::Class | :user | :object | :integer | ...`
- `:role => :concrete | :abstract` Concrete or virtual class (can make instances from it or not)
- `:pattern_match => :reactive | :nonreactive` Should change cause pattern-matching
- `:slots => Array | Hash` Contains slot list and with their options. In array variant you can specify only slot names, in hash you can override default attributes. Hash has structure `String => options` , where key is name of slot and options are:

- `:default => :derive | :none | String` Default value for slot
- `:default_dynamic => Boolean` Should be the default value dynamic or static
- `:storage => :local | :shared` Shared means that this slot is 'static' (shared between instances)
- `:access => :rw | :ro | :initialize | :read | :readwrite` Visibility of slot
- `:propagation => :inherit | :noinherit` Can the slot be inherited?
- `:source => :exclusive | :composite` More info in documentation
- `:pattern_match => :reactive | :nonreactive` Should change of slot cause pattern-matching
- `:visibility => :private | :public` Normal OOP visibility
- `:create_accessor => :none | :ro | :wo | :rw | :read | :write | :readwrite` Create default access function for this actions
- `:constrait => Constrait | Hash` Limitation of slot values

In instance acces method for each slot will be created, so it's important not to name slots after already defined methods. This method for slot return an instance of object `Class::Slot`, that have methods for changing slot options (named in same way as options in configuration hash).

Another instance methods:

- `new(Hash)` Return newly created instance (Clips::Instance) of this class, description is below when describing instancess of objects
- `instances()` Returns Array with all instances of this class
- `save()`
- `destroy()`
- `to_s()`

Example

```
# Playing with class
animal = Clips::Class.new :name => 'animal', :is_a => :user, :role => :abstract,
                  :slots => { 'age' => {:default => 0, :visibility => public}}
animal.save

dog = Clips::class.new :name => 'dog', :is_a => animal,
                  :slots => { 'name' => {}, 'race' => {:default => 'unknown'}}
dog.race.access = :readwrite
dog.save
```

4.2 Instances

New instance of class is created by calling method new(Hash) on it's Class (the capital letter is by purpose). As parametr it accept Hash with slot names as keys (both string or id is possible) and content as values that override default values.

- `[slot-name]` Read access method for slot (exist only if class declare 'create_accessor' at least for reading). After saving, it's synonym fo send get-slot message.

- `[slot-name]=` Write access method for slot (exist only if class declare 'create_accessor' for writing). After saving, it's synonym for send put-slot message.
- `duplicate(String, nil | Hash)` Returns copy of instance with new name and overrides given slots. If original instance is saved, than duplicated instance is saved as well. This copy is done by CLIPS function (duplicate-instance) and it's seems to be just a shallow copy.
- `initialize(nil | Hash)` Reinitalize object from it's defaults and slot overrides
- `send(String, *params)` Send a message to this object
- `save`
- `destroy`
- `to_s`

Example

```
# Creating with instances
puppy = dog.new :name => 'Lassie', :age => 0.2
puppy.race = 'Hasky'
puppy.save

# Duff is saved, bacause puppy is saved too
duff = puppy.duplicate('Duff', {:age => 1.0})
```

4.3 Message handlers

Message handlers lives next to instances and classes and are independent on them (just as they are in CLIPS). For this first version of scaffold, API is very simple and in fact and not much ruby-like (just wrapper about CLIPS code). Creation new message-handler is done by creating instance of MessageHandler class and saving it. As a pamarameters (in constructor and in access methods) accept strings that are directly inserted into CLIPS without any checks or any higner approach.

Constructor accept hash with values:

- `:name => String` Name of message handler
- `:type => :around | :before | :primary | :after` Type of message handler
- `:class => String | Class | :integer | ...` Class for handler
- `:params => String` Part of params (may be empty)
- `:body => String` Handler body (cannot be empty)

For every key in hash that constructor accept, instance have access (both read and write) method for changing its values (with same semantic) and in addition traditional set of methods:

- `save`
- `destroy`
- `to_s`

Example

```
# Creating message handler - so far ogly
hndl = Clips::MessageHandler.new :name => '+', :class => :numeric,
                :params => '?next', :body => '+ ?next ?self'
```

4.4 Calling ruby objects from CLIPS

rbClips is able to call ruby objects from CLIPS. There are two ways how to do it, manually and automatically. Automatic way is described later in section about rules. The manual way consist of two steps:

1. You have to save reference to object that you want to run throw `Clips::Base.save_reference(Object)`. Successfull calling return Fixnum that identify saved reference.
2. When you want call saved instance just use this fragment of CLIPS code: `(ruby ID method-name param-list*)`.

Please keep in mind, that I don't know ruby internals and it's Garbage collector. Code save only reference to object and don't know if that is sufficient for GB (so the object won't be removed from memory). In another words, don't pass here some local variable that will be deleted, but some global object, that remain in memory for all the time. Hopefully this limitation will be fixed later, when I start doing it.

4.5 Other methods for objects

Base object (Clips::Base) have same usefull wrappings for objects:

- `object_pattern_match_delayed(&block)` Run code in block with delayed pattern matching

COOL query system is accessible from Base object and consist of these six static methods:

- `any_instancep`
- `find_instance`
- `find_all_instances`
- `do_for_instance`
- `do_for_all_instances`
- `delayed_do_for_all_instances`

They share some commont settings, they all accept instance-set and query and some of them additionaly accept action to do. For this first scaffold, behaviour and parameters are simple. They accept hash and then generate the code:

- `:instance_set => String => String | Class | [ String | Class, ...]` Instance set, leading '?' in name (first string) is not compulsory, it will be added automatically if not present
- `:query => String` String with query (CLIPS code)
- `:action => String` Actions to do (CLIPS code)

Example

```
# Query system
male = Clips::Class :name => 'Male', :is_a => :user, :slots => ['age', 'name']
female = Clips::Class :name => 'Female', :is_a => :user, :slots => ['age', 'name']

Clips::Base.do_for_instances :instance_set => { 'human' => [male, female], 'female' => female},
         :action => "(+ ?human:age ?female:age)"
```

5 Facts

For working with Facts rbClips provides two classes Template, Fact.

5.1 Template

Entire workaround above ordered facts. Class is providing API for loading templates from CLIPS environment as well. Static methods:

- `new(Hash)` Descibed below, create new template object
- `load(String)` Load template form CLIPS and return it

Hash options for contructoruction of new template:

- `:name => String` Name of template in CLIPS world
- `:slots => Array | Hash` Slot list - in array accept only strings (names of slots). In hash variant accept String as key (name of slot) and another hash as value with options for this slot.
  - `:multislot => Boolean` Is this multislot? False by default.
  - `:default => :none | :derive | String` Default value for this slot
  - `:default_dynamic => Boolean` Should be default dynamic? Make sens only when some function is given as default value for slot.

Example

```
human = Clips::Template :name => 'human', :slots => ['name', 'age']
```

5.2 Facts

Clips::Facts class handle entire workaround above creating and deleting facts (both ordered and non-ordered).
Creation of new fact:

- `Clips::Fact.new(String, Array)` Create new ordered fact, Array can be blank
- `Clips::Fact.new(Template, Hash)` Create new non-ordered fact, Hash can be blank

Shared methods

- `template()` Return string in ordered fact and Instance of Template in non-ordered case
- `to_s()`
- `save()`
- `destroy()`

Ordered facts

- `slots()` Return array fact values
- `slots=(Array)` Redefine fact values

Nonordered facts

- `[slot-name]()` Return value stored in fact
- `[slot-name]=(Array)` Redefine value stored in fact

6 Rules

Rule class is almost most complex class in rbClips. It offers common static methods

- `new(Hash, &block)` Config hash, options are listed below, and block that set up the precodition and results.
- `load(String)` Load rule from CLIPS environment directly

Controller hash options:

- `:name => String` Rule name

6.1 Configuration block

Accept one variable that is used for build rule precondition (Left-hand side) and results (right-hand side) in one place. Given object of class Clips::Rule::Handler have methods for specifying left hand side and right hand side together because in most case LHS and RHS is connected as well. Ruby symbols means in Handler's methods CLIPS variable (e.g. :x will be transformed into ?x in CLIPS). There are two reserved symbols :one and :all that transforms into '?' and '$?'.

Simple pattern matching in LHS is done by pattern() method. It accept instances (not saved!) of matchable entities - facts and objects, with specified constraits on given slots or possitions. There are many of blank spaces that have to be hacked by givint string variable with CLIPS code (for example variable slot constraits - ?x&r̃ed&g̃reen). pattern() method also accept string with CLIPS fragment, where you can put whatever you want.

```
r = Clips::Rule.new :name => 'die' do |l|
  # For ordered facts
  l.pattern Clips::Fact.new 'human', [:all, 20] # Equals (human $? 20)
  l.pattern Clips::Fact.new 'human', [:one, 20] # Equals (human ? 20)
  l.pattern Clips::Fact.new 'human', [:x, 20]   # Equals (human ?x 20)

  # Nonordered facts
  l.pattern Clips::Fact.new humanfact, {:name => 'Honza', :age => :x}
    # Equals (humanfact (name Honza) (age ?x))

  # Objects
  l.pattern humanclass.new(:nick => 'Honza', :age => :x)
    # Equals (object (is-a humanclass) (nick Honza) (age ?x))

  # String hack
  l.pattern "(object (is-a X Y Z) (ahoj ?x:(numberp ?x)))"
```

```
end
r.save
```

You can also group conditions with 'and' and 'or' using blocks:

```
rule = Clips::Rule.new :name => 'some-rule' do |l|
  l.or do |ll|
    ll.pattern Clips::Fact.new 'human', [20]
    ll.pattern Clips::Fact.new 'human', [30]
  end
end
```

There are methods that modify both LHS and RHS - for example you searching for some pattern in fact and than you want to retract the fact - there are two operations (pattern matching and retraction) with one fact. In rbClips you specify this by only one method call `retract`, that have same parametres and behaviour as `pattern`, but additionally it add to RHS retract command.

```
rule = Clips::Rule.new :name => 'some-rule' do |l|
  l.retract Clips::Fact.new 'human', [20] # Search for it and than delete it
end
```

Calling ruby object as a result of rule (RHS) is simple just write

```
rule = Clips::Rule.new :name => 'some-rule' do |l|
  l.rcall ruby-instance, 'method-name', :a, 'x'
end
```

rcall method itself store reference to given instance, so you don't have to do it manually as was described earlier. When rule is fired, it run method 'method-name' from that object (instance) and give it two parametres - one is variable a that will be filled up with it's content (e.g. method will not receive symbal :a, but i will be substituted) and string 'x'.

You can specify more options to RHS using method `rhs(String)`, that add given fragment of CLIPS code to rhs with no changes.

Variable contraits You have to do it manually by giving fragment of CLIPS code.

7 Example of application

```
# Creating facts
Clips::Fact.new( 'animal', %w(dog) ).save
Clips::Fact.new( 'animal', %w(cat) ).save
Clips::Fact.new( 'animal', %w(horse) ).save
Clips::Fact.new( 'animal', %w(duck) ).save
Clips::Fact.new( 'child-of', %w(dog puppy) ).save
Clips::Fact.new( 'child-of', %w(cat kitten) ).save

# Creating rules
rule = Clips::Rule.new 'animalize'
rule.lhs do  |l|
        l.pattern Clips::Fact.new 'animal', [:animal]
        l.pattern Clips::Fact.new 'child-of', [:animal, :child]
end
```

```
rule.rhs = "(assert (animal ?child))"
rule.save

# Run
Clips::Base.run
```

8 Author's notes

- Design query system, something like (or whatever), just not to pass code directly in CLIPS
- Design blocks of code for query system - ruby blocks should be the best solution ;-)
- Make internal message buffer somehow prepared for stream (hack CLIPS to buindle it by changing main.o)