Rave Code Standards

Code & Modelling Standards

Göteborg February 2016

Table of Contents

About this Document

The purpose of this document is to create one unified guide for all implementation projects and customers to use and follow. As with any anything, common sense will always dictate how and when to apply the recommendation

RAVE Code Standard

Introduction

When writing code it is important to remember that someone else will likely be maintaining the code. This implies that you should write structured/readable code. Compact (terse) code may be elegant to write, but is generally more difficult to read and understand; this said, lengthy code lines can also be difficult to understand.

```
<-A printed page takes approximately 80 characters in one row->
```

Tab space standard is 4 characters. Naturally, when adding code to an existing CARMUSR, it is a good idea to remain consistent with the code already written. For all new implementations, use the guideline above.

Long expressions can be hard to indent and thus hard to debug. When reasonable, try to break up long expression into smaller calculations. An added benefit of this is that it will probably also increase the runtime performance of the code.

Comments in code

The purpose of writing code comments is to guide future users of your code, ultimately making maintenance easier. Also, file/module comments are intended to outline:

The purpose of the module (although this should be possible to generally infer from the module name itself.)

What rules, costs, definitions are in the module, especially the ones that are of a critical and/or complex nature.

What types of code should not be added to this module (where applicable)

Any other general information that will guide the developers as to how to maintain this module.

At the top of the file

```
/*
<module name>
------------------------------------------
  Purpose:
    Text describing the purpose…
   Sections:
```

```
    1. Bla bla
    2. Bla bla
 ---------------------------------------------
   Created by:
    Date  Name
   Major changes:
    Date  Name  Comment
 ********************************************/
module <name_of_the_module>
```

Separating comment

```
/*********************************************************
 * Section 1: Separation in the text to mark a new group
 * of definitions etc.
*********************************************************/
```

When separating different parts in the code use.

```
/*
** Rest rules
*/
```

Inline comment

Inline comments are comments on something special in the code.

```
/* Code comment because definition is complex! */
%apa% = %bepa% - 3:00;
```

Use inline comments with care.

Variables and functions

General Naming Convention

Variable naming is the most important mechanism for understanding the purpose of the code. The more explicit names used, the easier the code is to read.

```
%var_2% = %var_1% / 0:01;
%trip_minutes_int% = %trip_time% / 0:01;
```

In the above code snippet, `%var_2%` means very little unless you look at the code that defines it, whereas `%trip_minutes_int%` is explicit by looking at the variable name alone.

Rave definitions must always start with a letter in order to avoid problems when used in Python. For readability, all Rave code should be properly indented; short statements can be kept on one line, but longer ones should be spread over multiple lines. Rave does not care about whitespace, so each token in a statement can be split to different lines.

Prefixes/Suffixes

When defining variables and functions, a consistent use of prefixes and suffixes will further enhance the overall comprehension of the code. The following table outlines the basic set of suggested prefixes/suffixes:

Context	Description	Prefix	Suffix
Boolean	Any in context	any_	
	All in context	all_	
Dates/Times	Local	-	_lt
	UTC	-	_utc
	Homebase	-	_hb
	Scheduled	-	_sch
	Estimated	-	_est
	Actual	-	_act
Parameters	All	-	_p
Rules	Module name	rules_	
	Contract	contract_	_<level>_*

	General	general_	
Tables	Lookup variable/param		_table
Costs		cost_	
Penalties		pen_	
Max		max_	
Min		min_	

```
export %start_utc% = keyword_mappings.%departure%;
export %start_utc_sch% = keyword_mappings.%departure_sch%;
export %start_utc_est% = keyword_mappings.%departure_est%;
export %start_utc_act% = keyword_mappings.%departure_act%;
```

Identifiers

Rave is case-insensitive; `ThisVariableName`, `thisvariablename` or `ThIsVaRiAbleNAME` are all syntactically equivalent.

To keep things simple, and for runtime ease of use, all names should be written with a combination of lower case letters (`a-z`), and underscore (_) characters. Ultimately, this will facilitate searching for information in the rule set via 'grep' or 'show rule values'.

Variables

If the definition does not fit on one line, always break directly after '='.

```
%break_after_ok% =
    %eg_break_after_work% >= %break_after_long_work_min%;
```

Constants

Setting variables to constant values are often short, making it unnecessary to insert a line-break after the '='.

```
%break_after_min% = 1:00;


%generation_connection_time% =
    default(%trip_start%
              - prev(trip(roster),%trip_end%),
            0:10);
```

Parameters

If possible, write the entire statement on one row, otherwise move down and indent `parameter`, `remark` and `planner`.

Whenever sensible, use the `minvalue` and `maxvalue` options; this provides useful information to the user (concerning valid bounds of the parameter), as well as instructing the developer as to the intended limits for the given parameter.

Remarks

Parameters have a built in functionality to handle a one sentence remark that is presented to the planners. It explains what the parameter is for in a somewhat compact way. The remark should end with colon ‘:‘; this will make it look better when viewed in the parameters form in Studio.

```
%break_after_min_p% = parameter 1:00 remark "Remark: ";
```

```
%break_after_min_p% =
    parameter 1:00
    minvalue 0:30
    maxvalue 2:30
    remark “Enough to explain parameter in 45 chars: ",
    planner "A longer remark describing the parameter"
            " in more detail for the planner. This text"
            " can also be used to generate documentation";
```

The remark should provide some standard contextual information about the parameter; example, a parameter defining the name of an etable, it could begin with: “`Table`: “.

Never write a remark that is longer than 45 characters. In some windows in the system the complete remark will not be visible if you do so.

The specific role remarks, for example planner should be used to describe the rule in more detail. This remark should be used to describe the rule interpretation and if there are any specific information about the rule which is nice to know for the maintenance team etc. The system has the possibility to generate rule documentation from these remarks, which also makes it possible to create handover documentation etc.

Groups

When writing rules and variables, it is essential that code pieces that go together should be coded together (that is, in the same part of the module).

Grouping of variables and functions can be done in two ways, as a function group or as concept. The function group consists of rules and variables and parameters that are necessary to constitute a specific behaviour in the solution. This is the default grouping mechanism.

A concept group consists of definitions that are not necessarily related more than that they are different aspects of the same concept. An example is the definitions related to concept ‘start’ below. Even though the different definitions are not used by the same functions it is good to have them grouped together.

```
%start% = first(duty(trip), duty.%start%);
%start_date% = round_down(%start%, 24:00);
%start_tod% = time_of_day(%start%);
%start_tow% = time_of_week(%start%);
```

Order

Within a group a top-down definition strategy should be used as far as possible.

```
/* Code to calculate Apa. */
%apa% = %bepa% + 3:00;
%bepa% = %cepa% - %depa%;
%cepa% = 2 * %depa%;
%depa% = 0:01;
```

If-then-else expressions

Simple if-then-else

The standard way of writing an if-then-else block is in following form:

```
%standard_if% =
    if %check_this%
    then %use_this_variable% + %and_this%
    else %return_only_this_variable%;
```

Alternatively, the following for may be used:

```
%standard_if_alt% =
    if %check_this% then
        %use_this_variable% + %and_this%
    else
        %return_only_this_variable%;
```

Which of the styles to use is up to each implementation to decide, as long as there is good understanding and mixing of the two alternatives is kept to a minimum.

In general, code blocks should always be indented as shown above; however, with very short statements, you could write it all on one line:

```
%short_if% = if %_OK% then %_good% else %_bad%;
```

Nested if-then-else

Nesting of if-then-else expressions should be written:

```
%standard_if_then_else_if% =
    if %boolean_a%
    then %value_a%
    else if %boolean_b%
         then %value_b%
         else %value_c%;
or
%standard_if_then_else_if_alt% =
    if %boolean_a% then
        %value_a%
    else if %boolean_b% then
        %value_b%
    else
        %value_c%;
```

Other Nested if-then-else Expressions

```
%standard_nested_if% =
    if %boolean_a%
    then if %boolean_b%
         then %value_b%
```

```
          else if %boolean_d%
               then %value_d%
               else %value_e%
    else 0:00;
```

or

```
%standard_nested_if% =
    if %boolean_a% then
        if %boolean_b% then
            %value_b%
        else if %boolean_d% then
            %value_d%
        else
            %value_e%
    else
        0:00;
```

Logic expressions

There are different ways of building up logic expressions. The important thing to remember is that it should be easy to follow the logic. Rave evaluates logic expressions using lazy evaluation (once the condition is satisfied, or fails, the evaluation ends.) Place easy to calculate expressions first, as well as those that will catch the majority of cases; complex, less common expressions should be evaluated at the end.

Logic in if-then-else expressions

The `and` and `or` operands should be written first in a row if you intend to divide the row. This will make the logic/expression much easier to read.

```
%if_with_and% =
    if %boolean_a%
       and %boolean_b%
       and (%boolean_c1%
            or %boolean_c2%)
    then %value_a%
    else %value_b%;
```

As seen above, if the logical statement consist of both `and` and `or`, it is recommended to use parentheses.

Rules

A top-down definition strategy is used when defining rules; this means that the rule is defined first, followed by the parameters/variables that are used by the rule. Every rule should be written with a comment header:

```
/*
   Rule:
      <short rule name>

   Description:
      <description of rule functionality, assumptions etc>

   Agreement:
      <agreement ref to agreement documents and paragraphs>
*/
```

Short rule

```
rule legal_rule =
    valid %use_rule%;
    %actual_value% cmp %limit_value%;
    (%var_ok%;)
    startdate;
    failobject;
    failtext;
    remark "Control of parameter: ",
    planner "More descriptive information about the rule"
            " which can span for several lines.";
end

%use_rule% = parameter True;
```

Note Using a comparison in the rule itself, instead of a Boolean variable which contains the logic expression, this information can then be shown in check_legality and in rule_exception information.

Long rule

```
rule (off) legal_coupling_to_next_leg =
    valid %use_this_rule%
          and (%r_legal_liberal_coupling_control%
                 = %r_legal_liberal_coupling_control%);
    %coupling_to_next_leg% <= %max_time%;
    startdate;
    failobject;
    failtext;
```

```
    remark "LE_r3: Not more than 45 characters: ",
    planner "More descriptive information about the rule"
            " which can span for several lines.";
end
```

Failtexts cannot be longer than 70 characters (longer text will be truncated, with a warning being added to the logfile.)

Tables

Tables should be written in specific ways, depending on the usage context; example, tables used to lookup values for crew members should always be functions, whereas pairing attribute lookups can be normal variables.

Generic Tables

Generic Tables should be used, especially when retrieving data concerning crew in Crew Rostering, should always take a complete set of arguments for the lookup. Building access to tables in this way simplifies the integration of Crew Portal Bid filtering, as well as a number of scripting actions for automation purposes.

Bad

```
table crew_attributes =
    %id% -> String %main_func%;
    external %table%;
    CrewId -> MainFunc;
end
```

Good

```
table crew_attributes(String crewid) =
    crewid -> String %main_func%;
    external %table%;
    CrewId -> MainFunc;
end
```

Indentation is always an issue when defining tables; since tables can define many variables/functions, it is important to apply a sensible indentation scheme:

Short table

```
table shortchange_after_tab =
    flight_number -> int %shortchange_after%;
    external etab.%table_name_shortchange%;
    number -> number;
    - -> 0;
end
```

Long table (not very pretty, sometimes some groups can be made making it easier to count the columns):

```
table change_after_tab =
    flight_number,
    carrier,
    suffix,
    departure,
    arrival,
    %other_parameter% ->
        int %shortchange_after_tab%,
        int %longchange_after_tab%,
```

```
            int %middlechange_after_tab%;
        external etab.%table_name_longchange%;
        number, car, suff, dep, arr, an_parm
        ->
        shortnumber, longnummer, middlenummer;
        -,-,-,-,-,- -> 0,0,0;       /* This row should be compressed */
    end
```

Built-in Functions

When using built-in functions, and more specifically those taking multiple arguments, the code should be structured in the following way:

```
%generation_connection_time% =
    default(%trip_start%
             - prev(trip(roster),%trip_end%),
            0:10);
```

The arguments to the function are split in multiple lines, with the comma terminating the previous line.

Where statements

The standard formatting for short statements is:

```
%parm_1_sum% =
    sum(leg(trip), %parameter_to_add%) where (%add_up%);
```

If the statement is too long, you should break it up:

```
%parm_1_sum% =
    sum(leg(trip), %parameter_to_add%)
    where(%add_up%);
```

Best Practices

Performance

The performance of the rave code is the key to get performance both in optimization and in the editors. It is easy to degrade performance over time if not some thought is in place when writing rules, rudobs and other definitions.

The way to assure that degrading code is found is by using automatic tests, where generation speed in the optimizer is measured. The Rave Profiler can be used to show how time is spent when using a certain rule set. Measuring gui performance is harder and will in many cases include manual steps. At the beginning of a project, the generation speed should at least be 1500 rosters/second.

These test cases should then be executed ''before'' check in!

Memory

Rave is using its own internal memory when executed. For example, if you have made several definitions doing look-ups in one single table, Rave will load the same data several times. It is much better using a more complex function to do lookups, this will reduce the amount of memory needed.

Levels

The ruleset should define the minimum number of levels which are needed. The level definitions should be kept as simple as possible since the levels are the first things which are set up in rave. In the editor the levels are calculated again and again and again. Avoid using "while" and "where" in a level definition.

Dependency

All expressions in the language operate on the available level objects. In fact, an expression always belongs to and defines an attribute for a certain level. A flight’s departure and arrival times are attributes on the leg level and the sum of block times within a duty will be a duty attribute. The level to which an expression defines an attribute is called the dependency of the expression.

Rules and variables attributes should be of the correct level. They should not be evaluated on smaller objects than necessary.

Making a rule unnecessarily leg-dependent will cause Rave to check the rule on every leg. Mixing dependencies may also affect caching.

Range

The range of a variable or a rule is the set of level objects needed to evaluate it. For optimization runs, it is important that the range is as small as possible. If it is too large there will be a lot of unnecessary Rave evaluations. A leg rule or variable

with a range of chain is the worst case: each time the value is computed on a leg you need attributes from all legs in the chain.

Example:

```
/* Roster range */
%connection_time1% = next(leg(roster), departure) - arrival;
/* Duty range */
%connection_time2% = next(leg(duty), departure) - arrival;
```

The second definition in this example is much better, since Rave will only need to recalculate the connection time if the duty changes instead of every time the roster changes.

Caching

A very good principle is to try to re-use cached values as much as possible. Together with the dependency/scope principle will create efficient definitions where each calculation is done when it is needed and on the object it is needed.

Example bad performance

```
%duty_time_in_period%(Abstime a1, Abstime a2) =
    sum(duty(trip),
        duty.%duty_time_in_period%(a1,a2));
```

Example good performance

```
%duty_time_in_period%(Abstime a1, Abstime a2) =
    if %end% <= a1 or %start% >= a2
    then 0:00
    else if %start% >= a1 and %end% <= a2
        then %duty_time%
        else sum(duty(trip),
                duty.%duty_time_in_period%(a1,a2));
```

Iterator functions and caching

Try to avoid using `fundamental.%index%`, in the standard user, to get the index number of a times loop. Use the built-in `times_index(N)` instead. Rave is not able to cache any variable that depend on more than one index (`fundamental.%index%` depends on both `times_index(0)` and `index`). This does not apply to Rave code used for reports.

Lazy Evaluation

Logical expressions use lazy evaluation to improve performance. As soon as the outcome of an expression is fixed, the evaluation is stopped. For example:

`%a%` **or** `%b%`

If `%a%` is true, then we do not evaluate `%b%`.

Try to use it to your advantage. Examples of this is in if statements where an "easy" calculations should be put before a "heavy" one.

```
%my_expression_1% =
    if %easy_evaluation% or %heavy_evaluation%
```

```
then %value1%
else %value2%;
```

Let statements

Let constructs: Although it seems nice to use, these are never cached in Rave, so do not use it to *store* heavy calculations! ... instead move the definition to a separate variable.

Vertical Constraints

Vertical and global constraints should be avoided as much as possible. When you come across a vertical rule, try to see if it can be solved using other rules, introducing other optimization sequences etc.

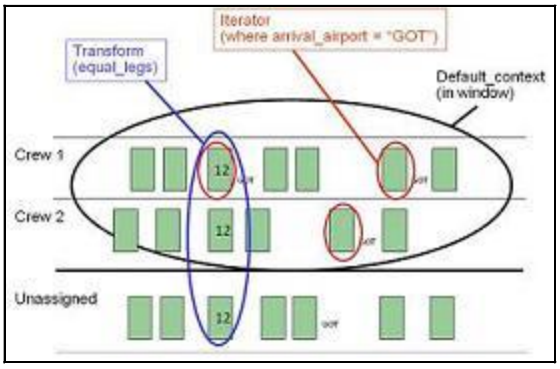
Contexts, Iterators and Transforms

Iterators divide a set of objects into smaller groups and sub-groups.

Contexts represents a certain group of objects (all trips in sub-plan, all objects in window…)

Transforms are entities containing all representations of a leg under specific considerations.

Iterators are for use mostly in reports and constraints. Context and complex iterators should be placed in the report code.



	Pairing Opt.	Rostering Opt.	Studio
Contexts			
sp_crew_chains		All rosters and unassigned trips in the optimizer	All rosters and unassigned trips in plan
default_context	All trips in the optimizer		All chains in window
sp_crrs			All trips in plan
Transforms			
equal_legs	Supported	Supported	Supported
equal_trips		Supported	Supported

Rules

Introduction

Rules are used to indicate an illegality in Studio or in the optimizer. A rule should depend on a comparison; this will make it easy to see what becomes illegal, both in the code itself, but also in studio and built in reports and functionality.

When creating rules it is very important to think about how the rule itself will be used and in which processes. When creating rules in pairing, you need to be aware of and code the rule making sure that it also works in a roster window if the code also will be used in rostering or tracking/cms. The same applies for rostering. All rules should have valid statements which inform the system where the rule is applicable.

If the rule is to be used in tracking/cms, also try to write rule so it is checked backwards. The standard has been to write rules looking ahead, but this is causing more work when it comes to coding understandable failobjects. There is no penalty in performance writing rules looking backwards in a chain.

It is also possible to introduce hierarchy in rules. Sub-rules may be grouped under a parent rule. The valid statement can be split between both; the remark string and failtext statements can be part of one or the other.

Valid Statements

All rules should have valid statements, and the easy to calculate statements should be the first ones to be evaluated! Rules should only be evaluated where they are needed. The rule itself can then assume that it is calculated in the right surroundings.

Parameter form / Group definitions

Rules and parameters in the system will not automatically be sorted or grouped in the parameter form available in Studio. They will by default be sorted the way they are defined in the source files (the order of the files themselves are not defined).

Within C14 and onwards, this is solved by using the new rave_group tag (instead of the old group tag). By using the `rave`_group tag, the parameters will appear in the order they are defined within the groupdefs file (and not based on the way they are defined in the source files). More information how this can be done can be found in the system documentation.

Rule Exceptions

The standard rule exception functionality is keyed on the rule-name and a time point. The rule exception needs to be modeled in each rule individually. If the rules are to be used in tracking/cms, always code the `startdate` and `enddate` tags and use scheduled times. If you are using the most-up-to-date times rule exceptions etc will be lost when the activity is moved in time.

If you write binary rules, they will not be able to use the "overshoot" mechanism which are available for comparison (<, <=, >, >=) rules.

Rule Documentation Strings

The rule remark should reflect the origin of the rule as far as possible. The remark is the text which is shown in Studio's parameter form. The '''planner''' or other documentation strings should provide a longer description on the rule. These texts can be used to generate rule documentation.

Rule Failtexts

The failtext are shown for the user when checking the rule illegalities in studio as well as when the users in looking in the Alert Monitor. The failtexts should be short. If you need more than 40 characters, try again!

Bad

```
"270 points rule failed forward maximum is 270 points, value is 273"
```

Good

```
"270p rule forward 273(270)"
"Open cabin time C173, 5 days, 2 FA, HEL"
"Service need exceeded by 2 in trip C173"
```

Some general guidelines when it comes to failtexts:

- Keep the text short, use abbreviations if possible
- Try to follow a standard for how to show text, value and limit
    - "text, value(limit)"
- Failtexts must not be dependent on the now time. This will make the AG update it in every loop.
- Failtexts longer than 45 characters will be truncated

Rule failobjects

Failobjects are used in tracking and also editors to display where the object which broke the rule is. It is important that the failobjects are correct; otherwise the information might be missed or ignored.

Independent on how the rule is written, the failobject must point out the activity where the rule is first broken. This general rule of thumb should cover 95% of all rules which is in a system. This is independent on if the rule is written to look forward, or backwards. In many cases there are much less work if the rules are written backwards when it comes to failobjects since then the system will default to the current object.

Note At the moment the system cannot display an illegality where there is no activity. Examples of rules which will generate a time only are recency rules etc.

In these situations you will need to choose a strategy like creating rudobs or visualize the illegality on the first activity assigned after the specific date.

If rules are written looking backwards there is a higher probability that the default failobject is correct, than if the rule is looking ahead. In all situations you as a developer need to check where the failobject is and modify it if it is not correct!

Limits

Adding a limit to a rule usually works well (e.g. "max consecutive..."), however, never have rules about "at least..." since in this case the solution starts as illegal and this will affect the performance.

Modelling

Introduction

Within Rave, it is possible to model things in many different ways, here are some best-practice on how things should be modelled to get them work in such an easy way as possible.

DST Secure

When an interval of any kind is going to be calculated from an object, always use the time-base UTC and NOT home-base or local-time. This will make the calculation safe and work properly even when the interval covers a DST change.

For example

```
rest_time =
    next(trip(wop), trip.%start_utc%) -
    trip.%end_utc%
```

Basic Time Definitions

When creating a Rave model where code will be used in planning as well as in day of operations it is very important that all code are written with some thought about time movement in mind. The following definition should be used in all levels!

`%start_utc%` is always the most up to date start time from scheduled, estimated and actual

`%start_utc_sch%` is always the scheduled start time, independent if it is delayed or moved forward

Accumulators

Accumulators are used to model rules spanning over long periods (historical/stored data and future/projection function). This can be done with accumulators:

- Evaluate: given a certain plan and stored data, evaluate the accumulator function for a particular interval.
- Accumulate data: given a certain plan and stored data, add the data of the current plan to the stored data.

Without accumulators, there is a need to manually read external tables and evaluate the difference between the elements to find out the accumulation for a certain period.