Run-time Checking, Testing, and Profiling for Cia

Table of Contents

1 Summary

This bundle implements a debugging, testing, and profiling framework (aka dynamic program analysis (`https://en.wikipedia.org/wiki/Dynamic_program_analysis`)) for Ciao.

As other components in Ciao (like documentation generator and static analyzer), this framework makes use of program assertions to obtain the specification of desired behaviors (e.g., test assertions).

It is integrated into the Emacs-based development environment (under the `CiaoDbg` menu) and the Ciao builder (`ciao test` command). See reference manual for more information.

2 Introduction

3 Run-time checking

Author(s): The Ciao Development Team.

Stability: [**beta**] Most of the functionality is there but it is still missing some testing and/or verification.

The `rtchecks` package provides a complete implementation of run-time checks of predicate assertions. The program is instrumented to capture violations of assertions at run time.

There are several main applications of run-time checks:

- To improve debugging of certain predicates, specifying some expected behavior that is checked at run-time with the assertions.
- To avoid manual implementation of run-time checks that should be done in some predicates, leaving the code clean and understandable.

3.1 Programs with assertions

There are two main entries in the main manual that explain the syntax and the semantics of assertions:

- see "⟨*undefined*⟩ *[The Ciao assertion language], page* ⟨*undefined*⟩" for general information on assertions and examples;
- see "⟨*undefined*⟩ *[Types and properties related to assertions], page* ⟨*undefined*⟩" for the formal definition of the syntax of several forms of assertions;

In Ciao assertions can be used to provide information on which properties should hold on predicate calls and successes. There can be several assertions for one predicate, e.g., in order to specify several possible calls and success patterns. Usually they are provided before the predicate definition in the source code:

```
% calculating length (as number of elements) N of a list L

:- pred len(L,N) : list(L)               => num(N) # "A1".
:- pred len(L,N) : (nnegint(N), var(L)) => list(L)# "A2".

len([],0).
len([_|T],N) :- len(T,M), N is M + 1.
```

The specification (assertions `A1` and `A2`) for the `len/2` predicate enumerates two possible ways for this predicate to be called:

- `A1`: if called with its first argument `L` a list (`basic_props: list/1`) then on success its second argument `N` should be a number (`basic_props: num/1`). Here, as no property is specified to hald on calls for the second argument `N`, the most general property (`basic_props: term/1`) is assumed for it;
- `A2`: if called with its second argument `N` a non-negavive integer (`basic_props: nnegint/1`), and the first argument `L` a free variable (`term_typing: var/1`), then on success its first argument `L` should be a list;

If no *assertion status* is specified explicitly, it is assumed that an assertion has a status `check`. By default, all assertions for a predicate with the `check` status are turned into run-time checks. This behavior may be modified either by setting Prolog flags (see ⟨*undefined*⟩ *[Other run-time checks configuration with flags], page* ⟨*undefined*⟩ below) or performing source code analysis in one of the predefined ⟨*undefined*⟩ *[Assertion checking levels], page* ⟨*undefined*⟩.

There are several cases in which a run-time check may fail:

1. a predicate was called with arguments that are outside of the acceptable call patterns (*calls check* failure); which can happen due to:

2. a predicate succeeded with argument values that are incompatible with the expected success pattern (*success check* failure), which can happen due to:
   1. a bug in the implementation of the predicate (most often cause);
   2. a too restrictive specification;

Below we illustrate each of the cases mentioned above.

Case 1: suppose that we call `len/2` with the wrong arguments, run-time checks will detect it and report this message:

```
?- len(a,z).
{In /tmp/foo.pl
ERROR: (lns 16-17) Run-time check failure in assertion for:
    foo:len(L,N).
In *calls*, unsatisfied property:
    list(L).
Because:
    ['L'=a].
ERROR: (lns 18-21) Failed in foo:len(L,N).
}

{In /tmp/foo.pl
ERROR: (lns 17-18) Run-time check failure in assertion for:
    foo:len(L,N).
In *calls*, unsatisfied property:
    nnegint(N).
Because:
    ['N'=z].
ERROR: (lns 18-21) Failed in foo:len(L,N).
}
```

Note how run-time checks report the first violated property: in the two error messages here for two `pred` assertions only one property violation is mentioned in each, although the `var(L)` from `A2` is clearly violated too.

However, this situation might also occur when the initial call is made with the acceptable arguments, but the computation is performed in a way that is not compatible with the specification:

```
?- len(L,1).
{In /tmp/foo.pl
ERROR: (lns 16-17) Run-time check failure in assertion for:
    foo:len(L,N).
In *calls*, unsatisfied property:
    list(L).
Because:
    ['L'=L].
ERROR: (lns 18-21) Failed in foo:len(L,N).
ERROR: (lns 18-21) Failed during invocation of len(_,_)
}

{In /tmp/foo.pl
ERROR: (lns 17-18) Run-time check failure in assertion for:
    foo:len(L,N).
In *calls*, unsatisfied property:
    nnegint(N).
```

```
Because:
    ['N'=N].
ERROR: (lns 18-21) Failed in foo:len(L,N).
ERROR: (lns 18-21) Failed during invocation of len(_,_)
}
```

Notice, how in the recursive clause of the `len/2` the unification for `M` happens after the evaluation of `len(T,M)` goal (due to the system-default left-to-right goal sequence reduction).

In this case we can re-write the `len/2` predicate, making it left-recursive and fully complying with its specification:

```
len([],0).
len([_|T],N) :- M is N - 1, len(T,M).
```

Case 2.1: imagine that incidentally a bug was introduced in the first clause of `len/2`:

```
len(_,0).
len([_|T],N) :- M is N - 1, len(T,M).
```

Run-time checks will detect that now `len/2` constructs not null-terminating lists:

```
?- len(L,1).
{In /tmp/foo.pl
ERROR: (lns 17-18) Run-time check failure in assertion for:
    foo:len(L,N).
In *success*, unsatisfied property:
    list(L).
ERROR: (lns 18-21) Failed in foo:len(L,N).
ERROR: (lns 18-21) Failed during invocation of len(_,_)
}

{In /tmp/foo.pl
ERROR: (lns 17-18) Run-time check failure in assertion for:
    foo:len(L,N).
In *success*, unsatisfied property:
    list(L).
Because:
    ['L'=[_|_]].
ERROR: (lns 18-21) Failed in foo:len(L,N).
}
```

Case 2.2: suppose the specification for `len/2` instead of `A2` contains `A2_too_strict`, which requires that on success `L` must be a list of integers, not just any terms.

```
:- pred len(L,N) : (nnegint(N), var(L)) => list(int,L). % A2_too_strict
```

Run-time checks will detect a failure of such assertion:

```
?- len(L,1).
{In /tmp/foo.pl
ERROR: (lns 17-19) Run-time check failure in assertion for:
    foo:len(L,N).
In *success*, unsatisfied property:
    list(int,L).
Because:
    ['L'=[_]].
ERROR: (lns 19-21) Failed in foo:len(L,N).
}
```

Bug: currently not all `rtchecks*` packages support interleaving assertions and predicate clauses. See `rtchecks/rtchecks_tr.pl` for details.

Note about performance: run-time checks usually affect the complexity of the program, which results in noticeable run-time overhead.

Consider a simplified specification of the `len/2` predicate:

```
:- pred len(L,N) : list(L) => num(N).

len([],0).
len([_|T],N) :- M is N - 1, length(T,M).
```

Checking that the first argument of the `len/2` predicate is a list at each recursive step turns the standard *O(n)* algorithm into *O(n^2)* one.

In Ciao the trade-offs between the thoroughness of the run-time checks and the associated run-time overhead are embodied into assertion checking levels. In the following we describe how to deal with the added complexity by run-time checks.

3.2 Assertion checking levels

There are several assertion checking levels availble in Ciao for different scenarios of run-time checking. They also represent different trade-offs between the number of run-time checks (overhead) and the respective code behavior guarantees (safety).

The `rtchecks` package provides two assertion checking levels, accessible through a package configuration flag:

- `rtchecks_level`
  - `exports`: Only use rtchecks for external calls of the exported predicates.
  - `inner` : Use also rtchecks for internal calls. Default.

The `rtchecks_opt` package provides three bacis assertion checking levels plus one optimization (see below):

- `unsafe`: no assertions are turned into run-time checks, zero run-time overhead added to program execution;
- `client_safe`: only assertions for the exported predicates are turned into run-time checks, the rest are ignored;
- `safe_rt`: all `check` assertions are turned into run-time checks;

Example of a module using run-time checks on one of the default checking levels:

```
:- module(ex_levels,[test/0,foo/2],[assertions,regtypes]).

:- use_module(engine(io_basic), [display/1,nl/0]).
:- use_package(library(rtchecks_opt/opt_rt_unsafe)).        % no checks
% :- use_package(library(rtchecks_opt/opt_rt_client_safe)). % interface
% :- use_package(library(rtchecks_opt/opt_rt_safe_rt)).     % all (interface + inter

test :- foo(1,X), bar(X,Y), display(Y), nl.

:- pred foo(A,B) : int(A) => atm(B) # "A1".

foo(1,a).

:- pred bar(C,D) : atm(C) => atm(D) # "A2".
```

```
bar(a,z).
```

On the `unsafe` checking level no assertions will be checked, on the `client_safe` only `A1` will be checked, and on `safe_rt` both `A1` and `A2` will be checked.

3.3 Check optimizations

3.3.1 Static checks optimization

As an optimization for the `safe_rt` assertion checking level, a variant of it, `safe_ctrt` level, is available:

- `safe_ctrt`: the program is subjected to static analysis pass, remaining `check` assertions after it are turned into run-time checks. In any case, `calls` assertions for exported predicates will always be turned into checks.

It is also available through the `rtchecks_opt` package:

```
:- use_package(library(rtchecks_opt/opt_rt_safe_ctrt)). % all + static analysis
```

If the `ex_levels` module from the example above was compiled on this checking level, the analysis would be able to prove `A2` to always hold, as well as the success part of `A1`, thus leaving only the calls part of `A1` for run-time checking:

```
%% %% :- check pred foo(A,B)
%% %%    : int(A)
%% %%   => atm(B).

:- check calls foo(A,B)
     : int(A).

:- true success foo(A,B)
     : int(A)
    => atm(B).

%% %% :- check pred bar(C,D)
%% %%    : atm(C)
%% %%   => atm(D).

:- true success bar(C,D)
     : atm(C)
    => atm(D).

:- checked calls bar(C,D)
     : atm(C).
```

More details are available from the related publication [SMH18b].

3.3.2 Shallow run-time checking

Note: this is a new feature and under active development. The documentation may be partial/obsolete.

Shallow run-time checking is an optimization of run-time checks of regular types in predicate calls across module boundaries in presence of hidden functors (via `:- hide` declaration of the `termhide` package).

Bug: this syntax is currently not working properly, please use this feature through an appropriate assertion checking level of `rtchecks_opt`.

To enable property caching in run-time checks use a variant of the `rtchecks` package, namely `rtchecks_shallow`, together with the `termhide` package:

```
:- module(MOD, _, [assertions,termhide,rtchecks_shallow]).
```

More details are available from the related publication [SMH18a].

The main reason for performing shallow run-time checking is run-time overhead reduction, especially in the cases where run-time checks change complexity of predicate execution:

```
:- module(bintrees,[root/2],[assertions,regtypes,nativeprops]).
%:- use_package(library(rtchecks_opt/shallow_rt_unsafe)).     % no checks
:- use_package(library(rtchecks_opt/shallow_rt_client_safe)). % interface
%:- use_package(library(rtchecks_opt/shallow_rt_safe_rt)).    % all (interface + inte
%:- use_package(library(rtchecks_opt/shallow_rt_safe_ctrt)).  % all + static analysis

:- hide(empty/0).
:- hide(tree/3).

:- regtype val_t/1.
val_t(X) :- int(X).

:- regtype tree/1.
tree(empty).
tree(tree(LC,X,RC)) :- tree(LC),val_t(X),tree(RC).

:- pred root/2 : tree * term => tree * val_t.

root(tree(_,X,_),X).
```

Here, the precondition checks for the `root/1` predicate change its complexity from constant to linear in the size of the input argument, as the checking code will need to traverse the input tree term to verify that it is a well-formed tree.

In presence of hidden functor terms, however, it might be possible to infer data shapes produced by the module, and simplify run-time checks w.r.t. this information.

```
:- regtype 'tree$shallow'/1.
'tree$shallow'(empty).
'tree$shallow'(tree(LC,X,RC)) :- term(LC),term(X),term(RC).
```

Note: currently it is not possible to print/output the inferred shallow regular types, as it is implemented as an internal compiler pass. However, they can be viewed at the WAM level. To produce the `MODULE.wam` expansion of the target module, in the Ciao shell do:

```
?- use_module(library(compiler)).
Note: module compiler already in executable, just made visible

yes
?- make_wam(MODULE).
```

The resulting shallow regular types for the `bintrees` module from the example above will look like this:

```
clause('bintrees:tree$shallow/1/1',
   [ifshallow
   ,neck(1)
   ,else
   ,endif
   ,get_constant_x0('bintrees:empty')
   ,proceed]).

clause('bintrees:tree$shallow/1/2',
   [ifshallow
   ,neck(1)
   ,else
   ,endif
   ,get_structure_x0(/('bintrees:tree',3))
   ,unify_x_variable(0)
   ,allocate
   ,unify_y_variable(1)
   ,unify_y_variable(0)
   ,init([])
   ,call(/('basic_props:term',1),2)
   ,put_y_value(1,0)
   ,call(/('basic_props:term',1),1)
   ,put_y_value(0,0)
   ,deallocate
   ,execute(/('basic_props:term',1))]).
```

3.3.3 Check caching

This is a new feature and under active development. This documentation may be partial/obsolete.

To enable property caching in run-time checks use a variant of the `rtchecks` package, namely `rtchecks_cached`:

```
:- module(MOD, _, [assertions,rtchecks_cached]).
```

More details are available from the related publication [SMH15].

Bug: currently checks only for a limited subset of predefined regtypes can be cached. See `rtchecks_cached/README` for details.

Example of use:

```
:- module(amqueue,[enqueue/3],[assertions,regtypes,rtchecks_cached]).
```

```
:- def_impl(var/1,fast_var/2).
:- def_impl(term/1,fast_term/2).
:- def_impl(pair_list/1,fast_listpair_cached/2).

:- pred enqueue/3 :  pair_list * term * var
              => pair_list * term * pair_list.

enqueue((Ins,Del),Elem,([Elem|Ins],Del)).

:- regtype pair_list/1.

pair_list((X,Y)) :- list(X), list(Y).
```

The `def_impl/2` declaration is needed at the moment to relate the regtype definitions in the source code and the corresponding checks with caching.

See `testsuite/rtchecks_cached/` for examples and benchmarks.

3.4 Run-time checking of `trust` assertions

By definition `trust/1` assertions are *trusted* by the compiler and their information is taken to be true. To detect cases where these assertions may be erroneous we allow an additional run-time checking level in which all assertions (including trusts) are checked. It is controlled by setting the appropriate Prolog flag:

- `rtchecks_trust`
  - `no` : Disable rtchecks for trust assertions.
  - `yes` : Enable rtchecks for trust assertions. Default.

3.5 Other run-time checks configuration with flags

> These flags are currently being reviewed and might not be documented/working as described here.

Run-time checks (specifically, those of the `rtchecks` package) can be configured using Prolog flags. Below we itemize the valid Prolog flags with their values and a brief explanation of the effects:

- `rtchecks_entry`
  - `no` : Disable rtchecks for entry assertions.
  - `yes` : Enable rtchecks for entry assertions. Default.
- `rtchecks_exit`
  - `no` : Disable rtchecks for exit assertions.
  - `yes` : Enable rtchecks for exit assertions. Default.
- `rtchecks_asrloc` Controls the usage of locators for the assertions in the error messages. The locator says the file and lines that contains the assertion that had failed. Valid values are:
  - `no` : Disabled.
  - `yes` : Enabled. Default.
- `rtchecks_predloc` Controls the usage of locators for the predicate that caused the run-time check error. The locator says the first clause of the predicate that the violated assertion refers to.

  - `no` : Disabled.
  - `yes` : Enabled, Default.
- `rtchecks_calloc`
  - `no` : Do not show the stack of predicates that caused the failure
  - `predicate`: Show the stack of predicates that caused the failure. Instrument it in the predicate. Default.
  - `literal` : Show the stack of predicates that caused the failure. Instrument it in the literal. This mode provides more information, because reports also the literal in the body of the predicate.
- `rtchecks_namefmt`
  - `long` : Show the name of predicates, properties and the values of the variables
  - `short` : Only show the name of the predicate in a reduced format. Default.

3.6 Custom property checks

This is a new feature and under active development. This documentation may be partial/obsolete.

In some cases declarative property definitions are not efficient enough to be used in the run-time checks instrumentation. In Ciao it is possible to write an alternative version of property that can be used specifically in run-time checks, while keeping the main version (which might be easier to read or is better understood by some static analyzer).

To provide the custom property implementation for run-time checking it is necessary to edit two files (suppose the original property is defined in a module `foo.pl`):

- `foo_rtc.pl`, module that contains the custom property implementation and is placed in the same folder as `foo.pl`. Make sure the custom property implementation is exported from both modules.
- `$CIAOROOT/core/lib/rtchecks/rtchecks_rt_propimpl.pl`, a database file that stores the links between different property implementations in the format of declarations '`:-rtc_impl(ModOr:PropOr/ArityOr, ModRt:PropRt/ArityRt).`' where `ModOr` and `ModRt` are names of the two modules that contain the original and the custom property definitions, `PropOr` and `PropRt` are the two different property implementations with respective arities `ArityOr` and `ArityRt`.

After these edits the `rtchecks` library needs to be rebuilt.

For an example of a system library that uses this feature see the `assertions/native_props` library.

3.7 Run-time checks implementation

During run-time checks transformation the following changes will be made to the source code:

- assertions will be expanded into respective checkable *calls* and *success* assertion conditions;
- for every instrumented predicate a wrapper clause will be generated;

Schematically, the run-time checks instrumentation looks as follows:

```
% original program source
:- pred P : Pre1 => Post1.
...
:- pred P : PreN => PostN.
```

```
P :- Body.
% instrumented program source
P :-    checkC(Pre1;...;PreN),
    Pnew,
    checkS(Pre1,Post1),...,checkS(PreN,PostN).

Pnew :- Body.
```

where:

- `checkC/1` is a simplified version of the *preconditions* check produced from the calls assertion condition, that checks if at least one calls pattern is satisfied in the predicate call, and emits a run-time error otherwise;
- `checkS/1` is a simplified version of the *postcondition* checks, produced from a success assertion condition, where each individual postcondition check is performed if the respective precondition held (with the possibility of emitting a run-time error if some property in a postcondition does not hold), or not performed if the precondition did not hold (and thus the postcondition is not applicable in the current predicate call).
- `P :- Pnew.` is the wrapper clause that maintains all original calls to the predicate `P` in the program, while redirecting them to `Pnew`, so that run-time checks are executed during the execution of `P`.

For testing instructions of a specific run-time checks implementation (packages `rtchecks`, `rtchecks_cached`, `rtchecks_opt`, and `rtchecks_shallow`) see the README files of the respective `testsuite/rtchecks*` directory.

3.8 Usage and interface

- **Library usage:**
  Add the `rtchecks` package to the module declaration as `:- module(...,...,[...,rtchecks]).` or include it explicitly as `:- use_package(rtchecks).`
- **Implicit imports:**
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

3.9 Other information

Note: the `assertions` package must always be included together with the `rtchecks` package (see `core/lib/compiler/global_module_options.pl` for details).

4 Unit testing

Author(s): Edison Mera, Pedro López, Manuel Hermenegildo, Alvaro Sevilla San Mateo, Nataliia Stulova, Ignacio Casso.

Stability: [**beta**] Most of the functionality is there but it is still missing some testing and/or verification.

The Ciao assertion language (see ⟨undefined⟩ [The Ciao assertion language], page ⟨undefined⟩) allows writing *tests* (including *unit tests*) by means of *test assertions*. These assertions make it possible to write specific test cases at the predicate level. This library contains predicates that can be used to run tests in modules and gather or pretty-print the results. It also provides some special properties that are convenient when writing tests and the corresponding run-time support.

4.0.1 Writing test assertions

As described in ⟨undefined⟩ [The Ciao assertion language], page ⟨undefined⟩ a *test assertion* is written as follows:

```
:- test predicate(A1, A2, ..., An)
   :  <Precondition>
   => <Postcondition>
   +  <Global properties>
   #  <Comment>.
```

Where the fields of the test assertion have the usual meaning in Ciao assertions, i.e., they contain conjunctions of properties which must hold at certain points in the execution. Here we give a somewhat more operational ("test oriented") reading to these fields:

- `predicate/n` is the predicate to be tested.
- `Precondition` is a goal (a literal or a conjuntion of literals) that is called before the predicate being tested, and can be used to generate values of the input parameters. While in some other types of assertions these preconditions contain properties to be checked, the typical role of the preconditions here is to provide concrete input values for which the predicate can be actually executed.
- `Postcondition` is a goal that should succeed after `predicate/n` has been called. This is used to test that the output of the predicate is the correct one for the input provided.
- `Golbal properties` specifies some global properties that the predicate should meet, in the same way as other assertions. For example, `not_fails` means that the predicate does not fail, `exception(error(a,b))` means that the predicate should throw the exception `error(a,b)`, and so on.
- `Comment` is a string that documents the test.

The following are some example tests for a complex number evaluator (see Chapter 8 [Examples (unittest)], page 31 for the full code):

```
:- module(ceval2, [ceval/2], [assertions, regtypes, nativeprops]).

:- test ceval(A, B) : (A = c(3, 4) + c(1, 2) - c(2, 3))
    => (B = c(2, 3)) + (not_fails, is_det).

:- test ceval(A, B) : (A = c(3, 4) * c(1, 2) / c(1, 2))
    => (B = c(3.0, 4.0)) + (not_fails, is_det).
```

```
ceval(A,   A) :- complex(A), !.
ceval(A+B, C) :- ceval(A, CA), ceval(B, CB), add(CA, CB, C).
ceval(A-B, C) :- ceval(A, CA), ceval(B, CB), sub(CA, CB, C).
ceval(A*B, C) :- ceval(A, CA), ceval(B, CB), mul(CA, CB, C).
ceval(A/B, C) :- ceval(A, CA), ceval(B, CB), div(CA, CB, C).

...

:- regtype complex/1.
:- export(complex/1).

complex(c(A, B)) :-
    num(A),
    num(B).
```

Test assertions can be combined with other assertions:

```
:- test ceval(A, B) : (A = c(3, 4) + c(1, 2) - c(2, 3))
    => (B = c(2, 3)) + (not_fails, is_det).
:- test ceval(A, B) : (A = c(3, 4) * c(1, 2) / c(1, 2))
    => (B = c(3.0, 4.0)) + (not_fails, is_det).
:- check pred ceval/2 : gnd * term => gnd * complex.
```

Test assertions can also take the standard assertion status prefixes. In particular, a status of `false` can be used to state that a test fails. This can be useful to flag bugs as known.

```
:- false test ceval(A, B) : (A = c(3, 4) + c(1, 2) - c(2, 3))
    => (B = c(2, 3)) + (not_fails, is_det).
```

Tests with a `false` (or `true`) prefix are not run.

There are some specific properties that only apply to testing which are provided in module `unittest_props.pl` (see Chapter 5 [Special properties for testing], page 23). For example, the limit to the number of solutions to be generated for the tested predicate can be set with the property `try_sols(N)`, a timeout to a test can be set with the property `timeout(N)`, `times(N)` specifies that the given test should be executed `N` times, etc.

4.0.2 Unit tests as examples

The special property `example` can be used to mark the unit test as an example, so that it is documented as such in manuals. The default behavior in `lpdoc` is to not include the unit tests in manuals unless they are marked this way. For example, the following test would be included in the manual as an example:

```
:- test ceval(A, B) : (A = c(3, 4) + c(1, 2) - c(2, 3))
    => (B = c(2, 3)) + (not_fails, is_det, example).
```

4.0.3 Running tests in a bundle

To run all these tests in a given bundle (as well as the other standard tests in the system) run the following (at the top level of the source tree or a bundle):

```
ciao test
```

4.0.4 Running the tests from the IDE

A convenient way to run these tests is by selecting options in the **CiaoDbg** menu within the development environment. This menu offers the following options:

1. `Run tests in current module`: execute only the tests specified in the current module.
2. `Run tests in current and all related modules`: execute the tests specified in the current module and in all the modules being used by it.
3. `Show untested exported predicates`: show the *exported* predicates that do not have any test assertions.

4.0.5 Running the tests from the top level or programmatically

The tests can also be run from the top level, loading this module (`unittest.pl`) and calling the appropiate predicates that it exports (see the module ⟨undefined⟩ [Usage and interface], page ⟨undefined⟩ section below). This can also be done from a program, provided it imports this module.

4.0.6 Combination with run-time tests

These tests can be combined with the run-time checking of other assertions present in the involved modules. This can be done by including the `rtchecks` package in the desired modules. Any `check` assertions present in the code will then be checked dynamically during the execution of the tests and can detect additional errors.

4.0.7 Integration tests

If you need to write tests for predicates that are spread over several modules, but work together, it may be useful to create a separate module, and reexport the predicates required to build the tests. This allows performing *integration testing*, using the syntax and functionality of the test assertions.

4.1 Usage and interface

- **Library usage:**
  `:- use_module(library(unittest)).`
- **Exports:**
  - *Predicates:*
    `show_untested_exp_preds/1`, `run_tests_in_dir_rec/3`, `show_test_output/2`, `show_test_related_output/2`, `run_tests/3`, `run_tests/4`, `run_tests_in_module/3`, `run_tests_in_module/2`, `run_tests_in_module/1`, `run_tests_in_module_check_exp_assrts/1`, `run_tests_related_modules/1`.
  - *Regular Types:*
    `test_option/1`, `test_action/1`.
  - *Multifiles:*
    `define_flag/3`, `test_filter/2`.

4.2 Documentation on exports

show_untested_exp_preds/1: PREDICATE

Usage: `show_untested_exp_preds(Alias)`

Show any exported predicates that do not have test assertions. This is an aid towards ensuring that all exported predicates have tests.

– *The following properties should hold at call time:*

`Alias` is a source name. (`sourcename/1`)

run_tests_in_dir_rec/3: PREDICATE

Usage: `run_tests_in_dir_rec(BaseDir,Opts,S)`

Executes all the tests in the modules of the given directory and its subdirectories. You can indicate that the modules in a sub-directory should not be tested by placing an empty NOTEST file in that sub-directory. Also, if a NOTESTFILES file is present, containing patterns for modules, those modules will not be tested. Unittest's statistical summary is summarised in S, which is given value 0 if there are as many successes as expected, and 1 otherwise.

– *The following properties should hold at call time:*

`BaseDir` is a pathname (encoded as an atom) (`pathname/1`)

`Opts` is a list of `test_option`s. (`list/2`)

`S` is a free variable. (`var/1`)

test_option/1: REGTYPE

A global option that controls the testing system. The current set of options is:

- `stdout(save)`: Save stdout as test output, do not show
- `stdout(dump)`: Save stdout as test output, then show
- `stdout(show)`: Just show stdout (do not save)
- `stdout(null)`: Ignore stdout
- `stderr(Opt)`: (same as above)
- `stderr(stdout)`: redirect stderr to stdout
- `dump_output`: Equivalent to `stdout(dump)`
- `dump_error`: Equivalent to `stderr(dump)`
- `rtc_entry`: Force run-time checking of at least exported assertions even if the run-time_checks flag has not been activated. (This is a workaround since currently we cannot enable runtime checks in system libraries smoothly).
- `treat_related` : Run tests in current and all related modules;
- `dir_rec` : Run tests in a specified directory recursively.

Usage: `test_option(Opt)`

`Opt` is a testing option.

test_action/1: REGTYPE

A global option that specifies a testing routine. The current set of actions is:

- `check` : run tests and temporarily save results in the auto-rewritable `module.testout` file;
- `show_output` : print the testing trace to the standard output;
- `show_stats` : print the test results statistics to the standard output;
- `save` : save test results file in `module.testout-saved` file;
- `briefcompare` : check whether current and saved test output files differ;
- `compare` : see the differences in the current and saved test output files in the diff format;

Usage: `test_action(Action)`

`Action` is a testing action

show_test_output/2: PREDICATE

Usage: `show_test_output(Alias,Format)`

Given a file `Alias`, tries to look up the respective unittest output file and print it to the standard output in the specified `Format` ('output' for test full trace, 'stats' for a statistical summary only, 'full' for both), otherwise emits a warning message that no test output file is avaiable.

– *The following properties should hold at call time:*

`Alias` is a source name. (`sourcename/1`)

`Format` is an atom. (`atm/1`)

show_test_related_output/2: PREDICATE

No further documentation available for this predicate.

run_tests/3: PREDICATE

No further documentation available for this predicate.

run_tests/4: PREDICATE

No further documentation available for this predicate. *Meta-predicate* with arguments: `run_tests(?,?,?,pred(1))`.

run_tests_in_module/3: PREDICATE

Usage: `run_tests_in_module(Alias,Opts,TestSummaries)`

Run the tests in module `Alias` (with options `Opts`). The results of the tests are returned as data in `TestSummaries`. `TestSummaries` can be pretty printed using `show_test_summaries/1` and `statistical_summary/2`.

– *The following properties should hold at call time:*

`Alias` is a source name. (`sourcename/1`)

`Opts` is a list of `test_option`s. (`list/2`)

– *The following properties should hold upon exit:*

`TestSummaries` is a list. (`list/1`)

run_tests_in_module/2: PREDICATE

Usage: `run_tests_in_module(Alias,Opts)`

Run the tests in module `Alias`. The results of the tests are printed out.

– *The following properties should hold at call time:*

`Alias` is a source name. (`sourcename/1`)

`Opts` is a list of `test_option`s. (`list/2`)

run_tests_in_module/1: PREDICATE

Usage: `run_tests_in_module(Alias)`

Run the tests in module `Alias` (using default options). The results of the tests are printed out.

– *The following properties should hold at call time:*

`Alias` is a source name. (`sourcename/1`)

run_tests_in_module_check_exp_assrts/1: PREDICATE

No further documentation available for this predicate.

run_tests_related_modules/1: PREDICATE

Usage: `run_tests_related_modules(Alias)`

– *The following properties should hold at call time:*

`Alias` is a source name. (`sourcename/1`)

show_test_summaries/1: (UNDOC_REEXPORT)

Imported from `unittest_summaries` (see the corresponding documentation for details).

statistical_summary/1: (UNDOC_REEXPORT)

Imported from `unittest_statistics` (see the corresponding documentation for details).

statistical_summary/1: (UNDOC_REEXPORT)

Imported from `unittest_statistics` (see the corresponding documentation for details).

get_statistical_summary/2: (UNDOC_REEXPORT)

Imported from `unittest_statistics` (see the corresponding documentation for details).

4.3 Documentation on multifiles

define_flag/3: PREDICATE

Usage: `define_flag(Flag,FlagValues,Default)`

– *The following properties hold upon exit:*

`Flag` is an atom. (`atm/1`)

Define the valid flag values (`flag_values/1`)

The predicate is *multifile*.

test_filter/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

4.4 Documentation on imports

This module has the following direct dependencies:

– *Application modules:*

`unittest_statistics`, `unittest_base`, `unittest_utils`, `unittest_db`, `unittest_regression`, `unittest_summaries`, `unittest_filters`, `unittest_runner`.

– *System library modules:*

`native_props`, `datafacts_rt`, `streams`, `stream_utils`, `terms`, `sort`, `aggregates`, `assrt_lib`, `system`, `hiordlib`, `c_itf`, `bundle_paths`, `lists`, `llists`, `system_extra`, `exemaker`, `source_tree`, `pathnames`, `messages`, `formulae`, `cpx_process`, `config_common`, `rtchecks_tr`, `write`, `apply_dict`.

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `hiord_rt`, `stream_basic`, `messages_basic`, `internals`, `runtime_control`, `system_info`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `isomodes`, `nativeprops`, `dcg`, `fsyntax`, `hiord`, `datafacts`, `define_flag`.

5 Special properties for testing

Author(s): Edison Mera, Pedro López, Manuel Hermenegildo.

Stability: [**beta**] Most of the functionality is there but it is still missing some testing and/or verification.

This module defines special properties and commands to be used in test declarations. They are called in general "test commands." This includes some that are random generators.

5.1 Usage and interface

- **Library usage:**
  `:- use_module(library(unittest/unittest_props)).`
- **Exports:**
  - *Properties:*
    `try_sols/2`, `times/2`, `generate_from_calls_n/2`, `timeout/2`, `near/3`, `user_error/2`, `test_command/1`, `bound_random/3`, `float_random/1`, `exp_float_random/1`, `exp_float_random_extended/1`.

5.2 Documentation on exports

try_sols/2: PROPERTY

Usage: `try_sols(G,N)`

For this test of `G` get at most `N` solutions (normally 2 solutions are generated, just enough to detect non-determinism).

- *The following properties hold globally:*
  `try_sols(G,N)` is a test command. (`test_command/1`)

Meta-predicate with arguments: `try_sols(goal,?)`.

times/2: PROPERTY

Usage: `times(G,N)`

This test of `G` should be repeated `N` times.

- *The following properties hold globally:*
  `times(G,N)` is a test command. (`test_command/1`)

Meta-predicate with arguments: `times(goal,?)`.

generate_from_calls_n/2: PROPERTY

Usage: `generate_from_calls_n(G,N)`

For this test of `G` generate (at most) `N` initial test states from the calls field (normally only the first solution is generated).

- *The following properties hold globally:*
  `generate_from_calls_n(G,N)` is a test command. (`test_command/1`)

Meta-predicate with arguments: `generate_from_calls_n(goal,?)`.

timeout/2: PROPERTY

Usage: `timeout(G,N)`

For this test of `G` abort if runtime exceeds `N` milliseconds (normally the default timeout is 600000 milliseconds, and can be altered using the 'unittest_default_timeout' flag). A timeout of 0 means no timeout

– *The following properties hold globally:*
`timeout(G,N)` is a test command. (`test_command/1`)

Meta-predicate with arguments: `timeout(goal,?)`.

near/3: PROPERTY

Usage: `near(A,B,Eps)`

Verifies that abs(`B - A`)/(abs(`B`) + abs(`A`)) =< `Eps`.

user_error/2: PROPERTY

Not implemented yet.

Usage:

The predicate should write to the current error stream the specified string.

test_command/1: PROPERTY

Usage: `test_command(X)`

`X` is a test command.

– *The following properties hold globally:*
`test_command(X)` is side-effect `free`. (`sideff/2`)

Meta-predicate with arguments: `test_command(goal)`.

bound_random/3: PROPERTY

Usage:

Similar to the predicate random:random/3

float_random/1: PROPERTY

Usage:

Similar to the predicate random:random/1

exp_float_random/1: PROPERTY

Usage:

Generates any floating point random number.

exp_float_random_extended/1: PROPERTY

Usage:

Generates any floating point random number including special cases like infinite, nan or zero whith sign.

5.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*
`random`, `aggregates`.

– *Internal (engine) modules:*
`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `hiord_rt`, `stream_basic`.

– *Packages:*
`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `hiord`.

6 Loading auxiliary test-related code

Stability: [**beta**] Most of the functionality is there but it is still missing some testing and/or verification.

This package provides declarations that can be used to load modules and packages needed for testing but which one does not want to be part of the module being tested.

6.1 Usage and interface

- **Library usage:**
  `:- use_package(unittestdecls).`
  or
  `:- module(...,...,[unittestdecls]).`
- **New declarations defined:**
  `load_test_module/1`, `load_test_module/2`, `load_test_package/1`, `unittest_default_timeout/1`.
- **Implicit imports:**
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `stream_basic`.
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

6.2 Documentation on new declarations

load_test_module/1: DECLARATION

Usage: `:- load_test_module(Module).`

Specifies an auxiliary module that must be loaded in order to execute the tests.

- *The following properties should hold at call time:*
  `Module` is a source name. (`sourcename/1`)

load_test_module/2: DECLARATION

Usage: `:- load_test_module(Module,PredNames).`

Specifies a module and the list of predicates that must be loaded in order to execute the tests

- *The following properties should hold at call time:*
  `Module` is a source name. (`sourcename/1`)
  `PredNames` is a list. (`list/1`)

load_test_package/1: DECLARATION

Usage: `:- load_test_package(Module).`

Specifies a package that must be used in order to execute the tests.

– *The following properties should hold at call time:*
`Module` is a source name. (`sourcename/1`)

unittest_default_timeout/1: DECLARATION

Usage: :- `unittest_default_timeout(Timeout)`.

Specifies a default timeout in millseconds for all the tests in the module. The value 0 means not timeout at all

– *The following properties should hold at call time:*
`Timeout` is an integer. (`int/1`)

7 Testing statistics

Author(s): Alvaro Sevilla San Mateo.

Stability: [**beta**] Most of the functionality is there but it is still missing some testing and/or verification.

This module implements predicates for generating statistical summaries for the testing processes.

7.1 Usage and interface

- **Library usage:**
  `:- use_module(library(unittest/unittest_statistics)).`
- **Exports:**
  - *Predicates:*
    `get_statistical_summary/2`, `print_statistical_summary/1`, `statistical_summary/1`.

7.2 Documentation on exports

get_statistical_summary/2: PREDICATE

Usage: `get_statistical_summary(IdxTestSummaries,Stats)`

Narrow the information of the tests and generate the statistical information structure needed to perform the statistical summary. `IdxTestSummaries` contains a list of terms with the results of tests.

print_statistical_summary/1: PREDICATE

Usage: `print_statistical_summary(Stats)`

Prints the statistical summary of the test, previously computed and given as input `Stats`.

statistical_summary/1: PREDICATE

Usage: `statistical_summary(IdxTestSummaries)`

Makes the statistic summary with the results of the tests. `IdxTestSummaries` contains a list of terms with the results of the tests.

- *The following properties should hold at call time:*
  `IdxTestSummaries` is a list. (`list/1`)

7.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*
`lists`, `llists`, `format`.

– *Internal (engine) modules:*
`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `messages_basic`.

– *Packages:*
`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

8 Examples (unittest)

Author(s): Edison Mera, Pedro López, Manuel Hermenegildo.

Stability: [**beta**] Most of the functionality is there but it is still missing some testing and/or verification.

Some examples of the use of test assertions.

8.1 test_examples.pl

```
:- module(test_examples,
    [
        p/1,
        display1/1,
        call_test10/1,
        cut_test5/0,
        display_fail/0
    ],
    [assertions, nativeprops, basicmodes, rtchecks, unittestdecls, hiord]).

:- doc(author, "Edison Mera").
:- doc(author, "Nataliia Stulova").

:- doc(module, "Some examples of unit tests.").

:- use_module(engine(io_basic)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% ----------------------------------------------------------------------
% tests only with preconditions

:- test p(A) : (A = c)
    # "test should pass".
:- test p(A) : (A = d, A = a)
    # "test should fail (precondition)".
:- test p(A) : throw(exception)
    # "test should fail (throws exception in precondition)".
:- test p(A) : (A = h)
    # "test should abort".

% ----------------------------------------------------------------------
% tests with preconditions and computational properties

:- test p(A) : (A = a) + not_fails
    # "test should pass".
:- test p(A) : (A = b) + fails
    # "test should pass".
:- test p(A) : (A = c) + exception(error(c, _))
    # "test should pass".
:- test p(A) : (A = c) + fails
    # "test should fail".
```

```
:- test p(A) : (A = c) + not_fails
    # "test should fail".

% ----------------------------------------------------------------------
% tests with preconditions and postonditions

:- test p(Z) => (Z = d).

% ----------------------------------------------------------------------
% assertions that can be added to all unit tests for p/1 with the
% 'rtc_entry' unittest option
:- check comp p(X) : (X = a) + not_fails.
% :- check comp p(X) : (X = b) + not_fails.

p(a).                               % rule succeeds
p(b) :- fail.                       % rule fails
p(c) :- throw(error(c, 'error c')). % rule throws an exception
p(h) :- halt(1).                    % rule aborts the execution

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- use_module(library(unittest/unittest_props), [times/2]).

% ----------------------------------------------------------------------
% this set of tests demonstrates the correct and incorrect uses of texec
% assertions, it only gives examples of passing tests
%
:- texec display1(A) : (A = hello)
    # "correct texecassertion, test should pass".
:- texec display1(A) : (A = hello) + times(1)
    # "correct texec assertion, test should pass".
:- texec display1(A) : (A = hello) + user_output("hello")
    # "incorrect texec assertion, test should pass".
% ----------------------------------------------------------------------

% ----------------------------------------------------------------------
% examples of the alternatives for the texec assertions above
%
:- test display1(A) : (A = hello)
    # "test should pass".
:- test display1(A) : (A = hello) + user_output("bye")
    # "test should fail".
% ----------------------------------------------------------------------

display1(A) :- display(A).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- test display_fail + (user_output("hello"), fails) # "Test OK".

display_fail :- display(hello), fail.
```

```
:- pred display_fail(A) + (user_output("hello2"), fails).
:- export(display_fail/1).

display_fail(_A) :- display(hello), fail.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- use_module(library(write), [write/1]).

:- test cut_test5 + (times(3), user_output("Cut disjunction"), fails) #
    "Test OK".

cut_test5 :- (! ; write('No')), write('Cut disjunction'), fail.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- test call_test10(X) : (X=(write(3), call(1)))
    + ( user_output("output"),
        exception(error(type_error(callable, 1), 'in metacall')) )
# "Wrong test".

:- meta_predicate call_test10(goal).

call_test10(X) :- call(X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- export(qs/2).
:- pred qs(+list(int), -list(int)) + is_det.

qs(A, A).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% tests below cover most of scenarios of the unit test expansion
%% (missing: tests that take into account different values of prolog flags
%%  rtchecks_predloc and rtchecks_asrloc (see rtchecks_tr:combine_locators/6)
%%  for the reason behind).

%% tests for different uses of GP properties in texec assertions

:- use_module(library(unittest/unittest_props), [try_sols/2]).

:- export(case_tx/2).

% processing texec as texec

:- texec case_tx(A, AB) : (A = a)
     # "test should pass".
:- texec case_tx(A, AB) : (A = a) + ( times(2), try_sols(2) )
```

```
     # "test should pass".

% processing texec as test

:- texec case_tx(A, AB) : (A = a) + is_det
     # "test should fail, bug: true&false test".
:- texec case_tx(A, AB) : (A = a) + (try_sols(2), non_det)
     # "test should pass".

case_tx(a, a).
case_tx(a, b).

% -----------------------------------------------------------------------

%% tests for different uses of AP and GP properties in test assertions

:- export(case_ts/2).

:- test case_ts(X, Y) : (X = a)
    # "test should pass".
:- test case_ts(X, Y) : (X = a) + try_sols(1)
    # "test should pass".
:- test case_ts(X, Y) : (X = b) + is_det
    # "test should pass".
:- test case_ts(X, Y) : (X = a) => (Y = b) + try_sols(1)
    # "test should fail".
:- test case_ts(X, Y) : (X = b) => (Y = a ; Y = b) + is_det
    # "test should pass".

% -----------------------------------------------------------------------

%% checks for the properties in the assertions are added to the unit
%% tests for case_ts/2 iff this module does not include the rtchecks
%% package. To see the effects of this assertions remove the rtchecks
%% package from the package list and run the tests for this module
%% wuth the [rtc_entry] test option. If you do not remove the
%% package, the assertions below will be simply turned into rtchecks.

% :- pred case_ts(X,Y) : atm(X) => int(Y) # "makes unit tests fail".
% :- pred case_ts(X,Y) : atm(X) => gnd(Y) # "does not affect unit tests".

case_ts(a, a).
case_ts(a, b).
case_ts(b, b).
```

8.2 ceval1.pl

```
:- module(ceval1, [ceval/2], [assertions, regtypes, nativeprops]).

:- doc(author, "Edison Mera").
```

```
:- doc(module, "This example illustrates how the unified approach for
   static verification, run-time checking, and unit testing works. It
   captures the situation in which the programmer changes the main
   functor of a data structure. In this case, the regular type complex
   c(A, B) is renamed to cm(A, B).  This file contains the initial
   program.").

:- entry ceval/2 : gnd * var.

:- regtype complex/1.
:- export(complex/1).

complex(c(A, B)) :-
    num(A),
    num(B).

:- test ceval(A, B) : (A = c(3, 4) + c(1, 2) - c(2, 3))
    => (B = c(2, 3)) + (not_fails, is_det).

:- test ceval(A, B) : (A = c(3, 4) * c(1, 2) / c(1, 2))
    => (B = c(3.0, 4.0)) + (not_fails, is_det).

:- check pred ceval/2 : gnd * term => gnd * complex.

ceval(A,   A) :- complex(A), !.
ceval(A+B, C) :- ceval(A, CA), ceval(B, CB), add(CA, CB, C).
ceval(A-B, C) :- ceval(A, CA), ceval(B, CB), sub(CA, CB, C).
ceval(A*B, C) :- ceval(A, CA), ceval(B, CB), mul(CA, CB, C).
ceval(A/B, C) :- ceval(A, CA), ceval(B, CB), div(CA, CB, C).

:- pred add/3 : complex * complex * var => complex * complex * complex.
add(c(A1, B1), c(A2, B2), c(A, B)) :-
    A is A1 + A2,
    B is B1 + B2.

:- pred sub/3 : complex * complex * var => complex * complex * complex.
sub(c(A1, B1), c(A2, B2), c(A, B)) :-
    A is A1 - A2,
    B is B1 - B2.

:- pred mul/3 : complex * complex * var => complex * complex * complex.
mul(c(A1, B1), c(A2, B2), c(A, B)) :-
    A is A1*A2 - B1*B2,
    B is A1*B2 + B1*A2.

:- pred div/3 : complex * complex * var => complex * complex * complex.
div(c(A1, B1), c(A2, B2), c(A, B)) :-
    D is A2*A2 + B2*B2,
    A is (A1*A2 + B1*B2) / D,
    B is (A2*B1 - A1*B2) / D.
```

8.3 ceval2.pl

```
:- module(ceval2, [ceval/2], [assertions, regtypes, nativeprops]).

:- doc(author, "Edison Mera").

:- doc(module, "This example illustrates how the unified approach for
   static verification, run-time checking, and unit testing works. It
   captures the situation in which the programmer changes the main
   functor of a data structure. In this case, the regular type complex
   c(A, B) is renamed to cm(A, B).  This file contains the renamed
   program (with a bug).").

:- entry ceval/2 : gnd * var.

:- regtype complex/1.
:- export(complex/1).

complex(cm(A, B)) :-
    num(A),
    num(B).

:- false test ceval(A, B) : (A = cm(3, 4) + cm(1, 2) - cm(2, 3))
    => (B = cm(2, 3)) + (not_fails, is_det).

:- test ceval(A, B) : (A = cm(3, 4) * cm(1, 2) / cm(1, 2))
    => (B = cm(3.0, 4.0)).

:- check pred ceval/2 : gnd * term => gnd * complex.

ceval(A,   A) :- complex(A), !.
ceval(A+B, C) :- ceval(A, CA), ceval(B, CB), add(CA, CB, C).
ceval(A-B, C) :- ceval(A, CA), ceval(B, CB), sub(CA, CB, C).
ceval(A*B, C) :- ceval(A, CA), ceval(B, CB), mul(CA, CB, C).
ceval(A/B, C) :- ceval(A, CA), ceval(B, CB), div(CA, CB, C).

:- pred add/3 : complex * complex * var => complex * complex * complex.
add(cm(A1, B1), cm(A2, B2), cm(A, B)) :-
    A is A1 + A2,
    B is B1 + B2.

:- pred sub/3 : complex * complex * var => complex * complex * complex.
sub(cm(A1, B1), cm(A2, B2), c(A, B)) :- % You forgot to rename c to cm!
    A is A1 - A2,
    B is B1 - B2.

:- pred mul/3 : complex * complex * var => complex * complex * complex.
mul(cm(A1, B1), cm(A2, B2), cm(A, B)) :-
    A is A1*A2 - B1*B2,
    B is A1*B2 + B1*A2.

:- pred div/3 : complex * complex * var => complex * complex * complex.
```

```
div(cm(A1, B1), cm(A2, B2), cm(A, B)) :-
    D is A2*A2 + B2*B2,
    A is (A1*A2 + B1*B2) / D,
    B is (A2*B1 - A1*B2) / D.
```

8.4 length.pl

```
:- module(length, [length/2, concat/3],
        [assertions, isomodes, metatypes, hiord, nativeprops]).

:- doc(author, "Alvaro Sevilla San Mateo").

length([],    0).
length([_|T], N) :- length(T, N0), N is N0 + 1.

concat([],     L,  L).
concat([X|L1], L2, [X|L3]) :- concat(L1, L2, L3).

:- test length(X, Y) : (X = [1, 2, 3], Y = 3) + not_fails.
:- test length(X, Y) : (X = [a, b, c, d, e], Y = 5) + not_fails.
:- test length(X, Y) : (X = [], Y = 0) + not_fails.
:- test length(X, Y) : (X = [a, b, c, d, e], Y = 5) + not_fails.
:- test length(X, Y) : (X = [a, b, c], Y = 5) + fails.
:- test length(X, Y) : (X = [a, b, c, d], Y = 5) + fails.
:- test concat(A, B, C) : (A = [1, 2], B = [3], C = [1, 2, 3]) + not_fails.
:- test concat(A, B, C) : (A = [1, 2], B = [3], var(C)) => (C=[1, 2, 3])
    + not_fails.
```

8.5 civil_registry.pl

```
:- module(civil_registry, [profession/1, surname/1, sevillian/1,
            family_income/1, married/2, salary/2],
        [assertions, isomodes, metatypes, hiord, nativeprops]).

:- doc(author, "Alvaro Sevilla San Mateo").

family_information(husband(name(antonio, garcia, fernandez),
            profession(architect), salary(300000)),
        wife(name(ana, ruiz, lopez),
            profession(teacher), salary(120000)),
        address(sevilla)).

family_information(husband(name(luis, alvarez, garcia),
            profession(architect), salary(400000)),
        wife(name(ana, romero, soler),
            profession(housewife), salary(0)),
        address(sevilla)).

family_information(husband(name(bernardo, bueno, martinez),
            profession(teacher), salary(120000)),
        wife(name(laura, rodriguez, millan),
```

```
            profession(doctor), salary(250000)),
        address(cuenca)).


family_information(husband(name(miguel, gonzalez, ruiz),
            profession(businessman), salary(400000)),
        wife(name(belen, salguero, cuevas),
            profession(housewife), salary(0)),
        address(dos_hermanas)).

profession(X) :- family_information(husband(_,profession(X),_),_,_).
profession(X) :- family_information(_,wife(_,profession(X),_),_).

:- test profession(X) : (X = doctor) + not_fails.
:- test profession(X) : (X = businessman) + not_fails.
:- test profession(X) : (X = computer_scientist) + fails.

surname(X) :-
    family_information(husband(name(_, X, _), _, _), _, _).
surname(X) :-
    family_information(_, wife(name(_, X, _), _, _), _).

:- test surname(X) : (X = alvarez) + not_fails.
:- test surname(X) : (X = salguero) + not_fails.
:- test surname(X) : (X = sanchez) + fails.

sevillian(X) :-
    family_information(husband(X, _, _), _, address(sevilla)).
sevillian(X) :-
    family_information(_, wife(X, _, _), address(sevilla)).

:- test sevillian(X) : (X = name(miguel, gonzalez, ruiz)) + fails.
:- test sevillian(X) : (X = name(belen, salguero, cuevas)) + fails.
:- test sevillian(X) : (X = name(pedro, sanchez, rodriguez)) + fails.
:- test sevillian(X) : (X = name(antonio, garcia, fernandez)) + not_fails.

family_income(X) :-
    family_information(
        husband(_, _, salary(N1)),
        wife(_, _, salary(N2)), _),
    X is N1+N2.

:- test family_income(X) : (X = 400000) + not_fails.
:- test family_income(X) : (X = 370000) + not_fails.
:- test family_income(X) : (X = 100) + fails.

married(X, Y) :-
    family_information(
        husband(name(X, _, _), _, _),
        wife(name(Y, _, _), _, _), _).
```

```
:- test married(X, Y) : (X = antonio, Y = ana) + not_fails.
:- test married(X, Y) : (X = miguel, Y = belen) + not_fails.
:- test married(X, Y) : (X = antonio, Y = maria) + fails.

salary(X, Y) :-
    family_information(husband(X, _, salary(Y)), _, _).
salary(X, Y) :-
    family_information(_, wife(X, _, salary(Y)), _).

:- test salary(X, Y) : (X = name(luis, alvarez, garcia), Y = 400000)
    + not_fails.
:- test salary(X, Y) : (X = name(laura, rodriguez, millan), Y = 250000)
    + not_fails.
:- test salary(X, Y) : (X = ana, Y = 100) + fails.
```

8.6 qsort.pl

```
:- module(_, [qsort/2], [assertions, nativeprops]).

:- doc(author, "Edison Mera").

:- doc(module, "This example illustrates how the unified approach
    for static verification, run-time checking and unit testing
    works. In this case, a bug have been introduced: in the call
    to append/3 by qsort/2, variables R1 and R2 are swapped.  Then
    we prove statically part of the assertions, for the part that
    cannot be verified statically we generate run-time checks,
    and finally, the program also has a test attached that
    exercises the run-time checks.").

:- prop sorted_num_list/1.
:- export(sorted_num_list/1).

sorted_num_list([]).
sorted_num_list([X]) :- number(X).
sorted_num_list([X, Y|Z]) :-
    number(X), number(Y), X<Y, sorted_num_list([Y|Z]).

:- calls partition(A, B, C, D) : (ground(A), ground(B)).
:- comp partition(A, B, C, D) : (list(num, A)) + (not_fails, is_det).
:- success partition(A, B, C, D) => (ground(C), list(num, D)).

partition([],    _B, [],        []).
partition([E|R], C,  [E|Left1], Right) :-
    E < C, !, partition(R, C, Left1, Right).
partition([E|R], C, Left, [E|Right1]) :-
    E >= C, partition(R, C, Left, Right1).

:- calls append(A, B, C) : (list(num, A), list(num, B)).

append([],    X, X).
```

```
append([H|X], Y, [H|Z]) :- append(X, Y, Z).

:- entry qsort(A, B) : (list(num, A), ground(A)).

:- test qsort(A, B) : (A = [5, 7, 2, 4, 3]) => (B = [2, 3, 4, 5, 7]).

:- calls qsort(A, B) : list(num, A).
:- comp qsort(A, B) : (list(num, A), ground(A), var(B)) + is_det.
:- success qsort(A, B) => (ground(B), sorted_num_list(B)).

qsort([X|L], R) :-
    partition(L, X, L1, L2),
    qsort(L2, R2), qsort(L1, R1),
    append(R2, [X|R1], R). % There is a bug here!!!
qsort([], []).
```

8.7 relationships.pl

```
:- module(relationships,
        [father/2, son/2, grandfather/2, brother/2, fsb/2],
        [assertions, isomodes, metatypes, hiord, nativeprops]).

:- doc(author, "Alvaro Sevilla San Mateo").

father('juan',   'maria'). % juan is the father of maria
father('pablo',  'juan').  % pablo is the father of juan
father('pablo',  'marcela').
father('carlos', 'debora').

% A is the son of B if B the father of A
son(A, B) :- father(B, A).

% A is the grandfather of B if A is the father of C and C is the father of B
grandfather(A, B) :-
    father(A, C),
    father(C, B).

% A and B are brothers if the father of A is also the father of B and A and B are di
brother(A, B) :-
    father(C, A),
    father(C, B),
    A \== B.


% A and B are fsb if A is the father, son, or brother of B
fsb(A, B) :-
    father(A, B).
fsb(A, B) :-
    son(A, B).
fsb(A, B) :-
    brother(A, B).
```

```
% Tests
:- test fsb(A, B) : (A = 'juan', B = 'marcela') + not_fails.
:- test fsb(A, B) : (A = 'pedro', B = 'pedro') + not_fails.
:- test fsb(A, B) : (A = 'debora', B = 'carlos') + fails.
:- test fsb(A, B) : (A = 'debora', B = 'nadie') + not_fails.
:- test brother(A, B) : (A = 'debora', B = 'carlos') + not_fails.
```

9 Profiler

Author(s): The Ciao Development Team, Edison Mera.

The Ciao profiler provides a high-level, flexible way to mark a predicate (or a literal) for profiling. This is done by using declarations to indicate if a program element must be instrumented or not.

The profiler module provides predicates and declarations to automatically handle profiling of the program to measure the time spent by predicates (or literals), as well as other cumulative statistics such as number of calls. It is made up of two parts: one implemented at the Ciao level, and another one implemented in C.

9.1 Profiling cost centers

By default, if the user does not specify anything, no predicate inside the module will be instrumented as cost center for profiling. The use of at least one declaration saying that a specific predicate must be instrumented overrides this behavior.

The declaration is as follows:

```
:- cost_center pred1/Arity1, ... predN/ArityN.
```

where `pred1/Arity1`, ..., `predN/ArityN` are the predicates to be instrumented as cost centers. They can be separated by commas or the can be in a list.

By default the engine hooks of all defined cost center are active. The declaration:

```
(predN/ArityN,nohooks)
```

will deactivate them.

Another useful declaration makes possible to indicate that a given predicate is not going to be instrumented as cost center:

```
:- no_cost_center pred1/Arity1, ... predN/ArityN.
```

where `pred1/Arity1`, ..., `predN/ArityN` are the predicates that will not be instrumented as cost centers. There are two options (as in the previous case): write one assertion for each predicate or declare more than one predicate (separated by commas or in a list) in only one assertion.

The following assertions define the behavior of all the predicates of the module:

```
:- all_cost_center.
```

```
:- all_no_cost_center.
```

They specify respectively that all the predicates in the module will be instrumented as cost centers and that no predicate in the module will be instrumented as cost center. In the first assertion the engine hooks of all defined cost centers are active. The declaration `:- all_cost_center(nohooks).` will deactivate them.

Cost centers can be also defined at literal level replacing the literal by the declaration:

```
:- cost_center(name_cc, literal)
```

where `literal` is the program literal and `name_cc` is the name of its associated cost center. At predicate level the name of both cost center and predicate are equal.

9.2 Usage and interface

- **Library usage:**
  Profiling requires compiling the engine with the `--debug-level=profile` or `--debug-level=profile-debug` options. For more information, see the `./ciao-boot.sh help`.
- **Implicit imports:**
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

10 Profiler utilities

This module provides predicates to interact with the profiler for particular calls.

10.1 Usage and interface

- **Library usage:**
  `:- use_module(library(profilercc/profiler_utils)).`
- **Exports:**
  - *Predicates:*
    `profile/1`, `profile_call/1`, `profile_reset/0`, `profile_info/1`, `activate_trace/0`, `deactivate_trace/0`, `activate_hooks/0`, `deactivate_hooks/0`.

10.2 Documentation on exports

profile/1: PREDICATE

Usage: `profile(Goal)`

Evaluates `Goal`, collect profile information related with the evaluation and dump the information.

- *Call and exit should be compatible with:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`cgoal/1`)

Meta-predicate with arguments: `profile(goal)`.

profile_call/1: PREDICATE

No further documentation available for this predicate. *Meta-predicate* with arguments: `profile_call(goal)`.

profile_reset/0: PREDICATE

No further documentation available for this predicate.

profile_info/1: PREDICATE

Usage:

- *The following properties should hold upon exit:*
  Term containing the information collected by the profiler (`profile_info_type/1`)

activate_trace/0: PREDICATE

No further documentation available for this predicate.

deactivate_trace/0: PREDICATE
No further documentation available for this predicate.

activate_hooks/0: PREDICATE
No further documentation available for this predicate.

deactivate_hooks/0: PREDICATE
No further documentation available for this predicate.

cost_center_ncnf/4: (UNDOC_REEXPORT)
Imported from `profiler_cc` (see the corresponding documentation for details).

cost_center_nf/4: (UNDOC_REEXPORT)
Imported from `profiler_cc` (see the corresponding documentation for details).

cost_center_nc/4: (UNDOC_REEXPORT)
Imported from `profiler_cc` (see the corresponding documentation for details).

cost_center/4: (UNDOC_REEXPORT)
Imported from `profiler_cc` (see the corresponding documentation for details).

cost_center/2: (UNDOC_REEXPORT)
Imported from `profiler_cc` (see the corresponding documentation for details).

total_time/1: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

have_overhead/2: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

dump_node_table_cc/0: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

do_profile_reset/0: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

set_hooks_active/1: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

get_hooks_active/1: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

set_trace_active/1: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

get_trace_active/1: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

get_trace_active/1: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

using_timestamp/1: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

profile_dump/0: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

cost_center_global_value/2: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

cost_center_edge_value/4: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

cost_center_node_value/3: (UNDOC_REEXPORT)
Imported from `profiler_utils_native` (see the corresponding documentation for details).

10.3 Documentation on imports

This module has the following direct dependencies:

– *Application modules:*

`profiler_rt`, `profiler_type`, `profiler_cc`, `profiler_utils_native`, `profiler_utils_base`.

– *System library modules:*

`native_props`, `read`, `stream_utils`, `system`.

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `io_basic`, `hiord_rt`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nativeprops`.

11 profiler_extra (library)

Implementation of some predicates related with the profiler, but that don't requires the activation of the profiling option in the engine.

11.1 Usage and interface

- **Library usage:**
  `:- use_module(library(profilercc/profiler_extra)).`
- **Exports:**
  - *Predicates:*
    `measure/2`, `measure/3`, `measure_nf/3`, `measure_0/3`, `get_option/1`, `time_option/2`.

11.2 Documentation on exports

measure/2: PREDICATE

Usage: `measure(Goal,Value)`

Same as measure/3, but uses ticks to measure the time.

- *The following properties should hold at call time:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`cgoal/1`)
- *The following properties should hold upon exit:*
  `Value` is a number. (`num/1`)

Meta-predicate with arguments: `measure(goal,?)`.

measure/3: PREDICATE

Usage: `measure(Option,Goal,Value)`

Unifies `Value` with the time spent in evaluate `Goal`, using the type of time `Option`.

- *Call and exit should be compatible with:*
  `Option` is currently instantiated to an atom. (`atom/1`)
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`cgoal/1`)
- *The following properties should hold upon exit:*
  `Value` is a number. (`num/1`)

Meta-predicate with arguments: `measure(?,goal,?)`.

measure_nf/3: PREDICATE

No further documentation available for this predicate. *Meta-predicate* with arguments: `measure_nf(?,goal,?)`.

measure_0/3: PREDICATE

No further documentation available for this predicate. *Meta-predicate* with arguments: `measure_0(?,goal,?)`.

get_option/1: PREDICATE

No further documentation available for this predicate. The predicate is of type *data*.

time_option/2: PREDICATE

No further documentation available for this predicate.

11.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`datafacts_rt`, `hrtime`.

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `hiord_rt`, `runtime_control`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `datafacts`.

12 Profiler auto-configuration of cost centers

Author(s): Teresa Trigo.

This module performs an automatic cost center configuration for a given goal. As a result it gives the trace of the highest resource (e.g., time, counts, ticks, etc.) consuming path starting from the goal.

12.1 Usage and interface

- **Library usage:**
  `:- use_module(library(profilercc/profiler_auto_conf)).`
- **Exports:**
  - *Predicates:*
    `cc_auto_conf/4`, `cc_auto_conf/5`, `cc_auto_conf/6`, `cc_auto_conf/7`, `cc_auto_conf/8`, `get_goal_filename/2`, `reset/1`, `record_time/1`, `tree_to_tex/2`, `recorded_time/1`, `write_cc_assertions/2`, `compose_name/3`.
  - *Multifiles:*
    `portray/1`.

12.2 Documentation on exports

cc_auto_conf/4: PREDICATE

Usage 1:

Same as `cc_auto_conf/5` applied to time.

- *Call and exit should be compatible with:*
  `Arg1` is a term which represents a goal, i.e., an atom or a structure. (`cgoal/1`)
  `Arg2` is a non-negative integer. (`nnegint/1`)
  `Arg3` is a list. (`list/1`)
  `Arg4` is any term. (`term/1`)

Usage 2:

Same as `cc_auto_conf/3` able to deal with modular programs.

- *Call and exit should be compatible with:*
  `info_item_name(Arg1)` (`info_item_name/1`)
  `Arg2` is a list. (`list/1`)
  `Arg3` is a term which represents a goal, i.e., an atom or a structure. (`cgoal/1`)
  `Arg4` is any term. (`term/1`)

Meta-predicate with arguments: `cc_auto_conf(goal,?,?,?)`.

cc_auto_conf/5: PREDICATE

Usage: `cc_auto_conf(Field,Goal,Cond,Goals,Tree)`

Auto configuration of cost centers in order to find the sub-graph `Goals` of the call graph that is responsible of the performance leak. `Goal` is the call to be profiled and `Field` is the resource name w.r.t. the optimization is made (e.g., time, counts, ticks, etc.).

– *Call and exit should be compatible with:*
`info_item_name(Field)` (`info_item_name/1`)
`Goal` is a term which represents a goal, i.e., an atom or a structure. (`cgoal/1`)
`Cond` is a non-negative integer. (`nnegint/1`)
`Goals` is a list. (`list/1`)
`Tree` is any term. (`term/1`)

Meta-predicate with arguments: `cc_auto_conf(?,goal,?,?,?)`.

cc_auto_conf/6: PREDICATE

No further documentation available for this predicate. *Meta-predicate* with arguments: `cc_auto_conf(?,?,goal,?,?,?)`.

cc_auto_conf/7: PREDICATE

No further documentation available for this predicate. *Meta-predicate* with arguments: `cc_auto_conf(?,goal,?,?,?,?,?)`.

cc_auto_conf/8: PREDICATE

No further documentation available for this predicate. *Meta-predicate* with arguments: `cc_auto_conf(?,?,goal,?,?,?,?,?)`.

get_goal_filename/2: PREDICATE

No further documentation available for this predicate. *Meta-predicate* with arguments: `get_goal_filename(goal,?)`.

reset/1: PREDICATE

Usage:

Given a name of a file erases its content

– *Call and exit should be compatible with:*
`Arg1` is an atom. (`atm/1`)

record_time/1: PREDICATE

No further documentation available for this predicate. *Meta-predicate* with arguments: `record_time(goal)`.

tree_to_tex/2: PREDICATE

No further documentation available for this predicate.

recorded_time/1: PREDICATE

No further documentation available for this predicate. The predicate is of type *data*.

write_cc_assertions/2: PREDICATE

Usage:

Given a set of predicates writes a cost_center assertion for all of them.

compose_name/3: PREDICATE

No further documentation available for this predicate.

12.3 Documentation on multifiles

portray/1: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

12.4 Documentation on imports

This module has the following direct dependencies:

- *Application modules:*

  `profiler_utils`, `profiler_base`, `profiler_extra`, `profiler_type`, `graph_to_tex`.
- *System library modules:*

  `native_props`, `foreign_interface_properties`, `datafacts_rt`, `aggregates`, `format`, `hiordlib`, `compiler`, `system`, `lists`, `llists`, `stream_utils`, `write`, `terms`, `c_itf`, `global_module_options`, `pretty_print`, `ciaoc_aux`.
- *Internal (engine) modules:*

  `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `hiord_rt`, `io_basic`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nativeprops`, `foreign_interface`, `basicmodes`, `regtypes`, `engine(foreign_types)`, `datafacts`.

13 Byrd box tracing model (library(tracing))

Author(s): Francisco Bueno.

This module can be used to spy predicates in a program. Spying a predicate means here that messages are displayed at the call, exit, fail and redo ports of the goals in the bodies of the clauses of the predicate spied.

In order to spy a predicate using this module a program expansion has to be performed on the program module defining the predicate. This is achieved by the `tracing` package.

This library is a programming exercise in CIAO expansion packages. It does not replace the standard interactive debugger, which has more functionality and is better integrated with the emacs mode. However, this library can sometimes be useful because it is more lightweight and allows tracing compiled code.

13.1 Usage and interface

- **Library usage:**
  `:- use_package(tracing).`
  in the program module defining the code to be spied; or
  `:- use_module(library(tracing/traces)).`
  to call the predicates exported by this module.
- **Exports:**
  - *Predicates:*
    `trace/3`.

13.2 Documentation on exports

spy/1: PREDICATE

`spy(M:F/A)`

Turns spying of predicate `F/A` in module `M` on. Declarations of the form `spy(F/A)` (where `M` is implicitly taken as the module where the declaration appears) can also be used, as long as the `tracing` package is included.

nospy/1: PREDICATE

`nospy(M:F/A)`

Turns spying of predicate `F/A` in module `M` off.

trace/3: PREDICATE

`trace(Goal,F,A)`

Displays messages at the ports of `Goal`, as long as spying is on for predicate `F/A` (module-name expanded). *Meta-predicate* with arguments: `trace(goal,?,?)`.

13.3 Documentation on imports

This module has the following direct dependencies:

- *Application modules:*

  `byrd`.
- *System library modules:*

  `datafacts_rt`.
- *Internal (engine) modules:*

  `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `hiord_rt`, `internals`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `hiord`, `datafacts`.

14 Byrd box tracing model

Author(s): Francisco Bueno.

This module can be used to spy predicates in a program. Spying means here to display messages at the call, exit, fail, and redo ports of goals for the predicate spied.

In order to spy a predicate using this module a program expansion has to be performed on the program module defining the predicate. This is achieved by the `byrdbox` package.

This library is a programming exercise in Ciao expansion packages. It is mainly intended as an auxiliary library for `library(tracing)`.

Please note that this is not a replacement for the standard debugger (and which can also be embedded in executables), since it has more limited functionality.

14.1 Usage and interface

- **Library usage:**
  `:- use_package(byrdbox).`
  in the program module defining the code to be spied; or
  `:- use_module(library(byrdbox/byrd)).`
  to call the predicates exported by this module.
- **Exports:**
  - *Predicates:*
    `spy/1`, `nospy/1`, `trace/1`, `trace0/1`.

14.2 Documentation on exports

spy/1: PREDICATE

`spy(M:F/A)`

Turns spying of predicate `F/A` in module `M` on. Declarations of the form `spy(F/A)` (where `M` is implicitly taken as the module where the declaration appears) can also be used, as long as the `byrdbox` package is included.

nospy/1: PREDICATE

`nospy(M:F/A)`

Turns spying of predicate `F/A` in module `M` off.

trace/1: PREDICATE

`trace(Goal)`

Displays messages at the ports of `Goal`, as long as spying is on for its predicate. *Meta-predicate* with arguments: `trace(goal)`.

trace0/1: PREDICATE

`trace0(Goal)`

Displays messages at the ports of `Goal`. *Meta-predicate* with arguments: `trace0(goal)`.

14.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`, `messages`.
- *Internal (engine) modules:*
  `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `hiord_rt`, `internals`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `datafacts`.

References

[SMH15] N. Stulova, J. F. Morales, and M. V. Hermenegildo.
Practical Run-time Checking via Unobtrusive Property Caching.
Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15) Special Issue, 15(04-05):726–741, September 2015.
http://arxiv.org/abs/1507.05986.

[SMH18a] N. Stulova, J. F. Morales, and M. V. Hermenegildo.
Exploiting Term Hiding to Reduce Run-time Checking Overhead.
In Francesco Calimeri, Kevin Hamlen, and Nicola Leone, editors, *20th International Symposium on Practical Aspects of Declarative Languages (PADL 2018)*, number 10702 in LNCS, pages 99–115. Springer-Verlag, January 2018.

[SMH18b] N. Stulova, J. F. Morales, and M. V. Hermenegildo.
Some Trade-offs in Reducing the Overhead of Assertion Run-time Checks via Static Analysis.
Science of Computer Programming, 155:3–26, April 2018.
Selected and Extended papers from the International Symposium on Principles and Practice of Declarative Programming 2016.

Library/Module Index

B

P

R

T

U

Predicate Index

Property Index

Regular Type Index

Declaration Index

L

U

Concept Index

Author Index

Global Index

This is a global index containing pointers to places where concepts, predicates, modes, properties, types, applications, authors, etc., are referred to in the text of the document.

$

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Y