LeadsTo Software

Lourens van der Meij
E-mail: lourenstcc@gmail.com

June 15, 2016

Abstract

This document describes the LeadsTo software in detail. It starts out as an investigation into details of the algorithm.

Contents

1 LeadsTo core design and implementation

1.1 Introduction

A leadsto specification is encoded as a pl file containing predicates.

1.2 run_simulation/2

Here we describe the main leadsto procedure `run_simulation/2`, `run_simulation(File, Frame)` This leadsto specification *File* is loaded into the leadsto runtime:

1. The predicates are loaded into module `spec`.
2. After that, all terms in the input file are preprocessed, often leading to asserted dynamic predicates in the current(algo) module.

   There seems to be almost no compilation at this stage and it looks like terms in thee spec module often are asserted as facts into algo without any transformation.

   Some translation of sortdefs is performed. If a sort contains less than 100 ground terms, it is instantiated, otherwise the sort definition is left as is. This leads to spec:sortdef(Sort, Terms). The source contains a beginning of a new way of encoding sort definitions.
3. If the specification contained a model specification, we run each model instance after setting up model parameters. Otherwise we perform a single run.
4. Running the specification
   (a) setup of the runtime (first part of `runspec1/0`)
   (b) performing the firing of rules (`runspec_rest/0`)
5. Saving the generated trace

1.3 LeadsTo specifications

1.3.1 Sources for information

The file `userman.html` contains the documentation for most allowed constructs in LeadsTo specifications.

The file `olddoc/syntax.txt` also describes the syntax. See section 6.

2 Details

2.1 Times: Handled Time, Setup Time, Start Time, End Time

setuptime, `TSetup` In practice identical to `TStartup`, but there are options for defining `TSetup` to have a value before `TStart` so that leadsto rules could fire for antecedent values before

TStart and use cwa(Atom) derived values to make them fire. The current value of TSetup is stored in dyn_setup_time(TSetup).[1]

handledtime Handled time is initialized by setup_unknown_or_cwa/2 to TStart. There is an additional implied condition on handled time: "You should never ask for values before TSetup" and "All atoms that have no explicit trace entry before THandled have value false if cwa, unknown otherwise".

starttime, TStart Start time. The algorithm uses TSetup, but TStart still plays a role, when storing traces, only values at/after TStart are saved.[2] TStart is stored as dyn_start_time(TStart), but only called through start_time/1.

endtime If not specified there currently is a default of 200 (see end_time/1)

TSetup and TSetup are set up in do_setup_time(TStart, TSetup). They may contain specification constants.[3]

HandledTime is incremented in handled_time_step/1 and runspec_rest/0 ensures that at the end of the leadsto algorithm HandledTime >= Endtime.

2.2 Traces

2.2.1 Datastructures

Traces are stored as Prolog facts, each fact represents values of a single ground atom.

Values of an atom over time are represented as a list:

```
[range(23.0, 24.0, true), range(17, 18, true)]
```

The ranges are ordered, the latest time range first. During the execution of our leadsto algorithm, only necessary values are stored, unknown ranges or false ranges where cwa(Atom) holds are not part of the trace.

In saved traces all values are represented. Saved traces will only contain atom values in the range start_time to end_time. The leadsto algorithm may derive values outside of that range.[4]

2.2.2 Algorithm Variables

dyn_sim_status(*File, Status*)
says in what stage of loading and running the algo algorithm is. *Status* is loaded, running or done.

[1] We had command line options setup_maxg and setup_maxfg for that purpose that would introduce TSetup based on the maximum leadsto rule values for *f* and *g*. But the sourcecode says this is not supported. So, in practice TStart == TSetup

[2] TODO:check this!

[3] In do_setup_times/2 the values are passed through tr_basic_element(Term, [], TermOut) that will substitute spec:constant(Name, Val) occurrences. See section 3.1.

[4] TODO: We should discuss alternatives:

1. Why not save only necessary values in saved traces?
2. If saving everything, why not compact the timerange? [range(T1, T2, TFU1), range(T2, T3, TFU3), ...] into [T1-TFU1, T2-TFU2, .. TE-[]]

dyn_currently_loaded(*Kind, File*)
says what *File* is loaded and what *Kind*, where *Kind* is `trace` or `sim`.

2.2.3 Loading and Trace Generation

load_simulation(+*File*)
The specification *File* is loaded into module `spec`. The source code seems somewhat complex: The module `spec` is set up, `discontinous/1` directives are generated for all leadsto specification terms, the terms are read from *File* and asserted one by one into the `spec` module.

Command line constants are added to module spec (see section 3.1).

Then, the leadsto specification is read one more time, and each `Term` is passed on to `handle_term/1`. Most terms are handled by asserting dynamic facts into module `algo`. Some of those are 1-1 translations, others are not.

`model(Model)` is translated into `dyn_model(Model)`, after checking that there is only one such term.

`cwa(F/A)` is translated into `dyn_cwa(FunctorTerm)`.

The most interesting things happen with interval leadsto specification terms and leadsto rule terms. Interval rules are converted into two standard forms and stored by `initialise_interval/3` and `initialise_interval_periodic/4`.

`handle_term/1` itself simply assertz leadsto rules into `dyn_leadsto(RuleId, Vars, Antecedent, Consequent, Delay)` facts. `setup_leadsto/6` processes these facts further in `runspec1/0` at algorithm startup.

Finally `update_sorts/0` performs some pre compilation of sort definitions. [5]

initialise_interval(+*Range*, +*Vars*, +*LiteralConjunction*)
Handles initial setup of all non-periodic interval rules. In this phase the predicate asserts `dyn_interval(i(Range, Vars, LiteralConjunction)`. `setup_rt_intervals/0` processes the terms further at algorithm startup.

initialise_interval_periodic(+*Range*, +*Period*, +*Vars*, +*LiteralConjunction*)
the predicate asserts `dyn_interval(i(Range, Period, Vars, LiteralConjunction)`. `setup_rt_intervals/0` processes the terms further in `runspec1/0` at algorithm startup. `initialise_interval/3` and `initialise_interval_periodic/4` deal with them.

Leadsto rule terms are translated into `dyn_leadsto(I, Vars, LitDisConj, AndLiterals, Delay)` facts.

---

[5] TODO: It is unclear what happens to sortdefs at this stage. Questions are:

- What terms are used? Are they spec:sortdef/2 and spec:sortdef/4?
- What is the role of `dyn_sortdef/5`?
- Are constants somehow substituted into sortdef elements?

Question surrounding `load_simulation/1`:

- Why is the leadsto specification scanned twice. The terms will be probably be walked through even one more time to set up the algorithm.

load_simulation/1 sets dyn_sim_status(File, loaded).

reset_sim_info
clears the content of spec together with other run time information.

runshowspec(+*Frame*)
Two parts, runspecdo/1 and show_results/2.

runspecdo(+*Frame*)
(Functionality in runspec/1). If we are dealing with a model, we initialize the output trace common to all model traces, then for each model instantiation we call runmodel/4 that does runspec1/0 and cleans up after itself for the next runmodel/4.

If there is no model/1, we call runspec1/0 followed by savetrace/1.

runspec1
This procedure calls do_setup_time/2 that sets up TSetup and TStart.

In runspec/1 we perform: setup_rt_intervals/0, setup_unknown_or_cwa/2, setup_leadsto/6 for each leadsto rule dyn_leadsto/5, get_model_checking_p_rules/0 (??), setup_atom_state_boundaries/0 and finally do the real reasoning in runspec_rest/0.

setup_rt_intervals
For every dyn_interval/1 term we perform init_interval_callbacks/10 where all but the first three arguments are callback variables or callback predicates.

The setting of the interval rules does some detailed steps such as variable instantiation. Finally this leads to changes to dyn_atom_trace/3. See

```
initialise_interval_p(Interval, P, Vars, Form1) :-
        (       instantiate_vars(Vars, VarsInst),
                tr_range(Interval, VarsInst, T1, T2),
                tr_basic_element(P, VarsInst, P1),
                ...
```

setup_leadsto(+*TStart*, +*Vars*, +*Antecedent*, +*Consequent*, +*Delay*, +*RId*)
(Called by runspec1/0). It translates and transforms the rule using init_interval_callbacks/10, then calls setup_lt_internalL/5 which simply calls setup_lt_internal/6. That predicate simplifies Antecedent and Consequent terms using simplify_term/4. In some cases the rules simplify to specifying constant Antecedents. Other cases are passed on to setup_nontrivial_leadsto/5. It stores the compiled leadsto rule as dyn_lt_rule(Id, AnteLits,ConseRId,PVOutC,Delay,RId) but also does initial firing and instantiation the leadsto rule data structures by calling setup_lt/6 which calls setup_lt_normed/8 which is part of the run time algorithm.

init_interval_callbacks(*TmInf, Vars, Forms, TmInf1, Vars1, Forms2, InvldVars, InvldTimeInfo, ActPreInsttiated*)
is used for setting up interval rules and leadsto rules. It instantiates variables, also takes care of forall/2 terms(instantiates them).

Traces are generated by the main algorithm in module algo. They are internally stored as `dyn_atom_trace(AtomKey, Atom, AtomTrace)` facts. For performance reasoning traces that can no longer play a role in the algorithm are backed up into `dyn_atom_trace_backup/3` facts. [6]

2.2.4 Saving traces

Traces are saved in two stages by

savetracesetup(+*File*, +*Frame*, -*Telling***)**
Saves constants and sets up trace storage stream.

savetrace1(+*TraceName***)**
Saves the trace itself. (If *TraceName* is `[]`, trace will not have trace id.)

savemodelspec_cleanup(-*TellStream*, +*ModelInstanceTraceName***)**
If the *leadsto specification* contains a model, the separate model instances saved.

3 Details

3.1 Constants

One can define *specification constants* constants that will be substituted into leadsto specification elements. Within a leadsto specification we use:

```
constant(Name, Value).
```

From the command line one can specify `-constant Name=Value`. This adds a constant to the specification. Value must be a valid ground Prolog term.

`set_option_constant/1` handles this by asserting `dyn_add_cmd_constant/2`.

util:load_cmd_constants/0 loads those constants into module `spec` as `constant(Name, Value)` facts.

Constants are substituted by the procedure `tr_basic_element(Term, [], TermOut)`. Constants are stored as `spec:constant(Name, Val)` values.

3.2 Model Checking

The source contains code labelled *model checking*. I do not remember whether this code ever worked. I seem to remember I tried converting the leadsto model into some state based form.

`Makefile` contains an example call of using modelchecking:

```
./leadsto -local -modelchecking spec/heartn.lt
```

The only visible result seems to be some debugging info on the screen.

A first look at the code in *modelchecking.pl* does not make anything clear yet.

There is a document `olddocs/modelchecking.doc` that may provide background. I fear that the code that is still present in `algo.pl` never really did anything.

[6] TODO:I seem to remember that at places in the algorithm we depend on there either being `dyn_atom_trace/3` or `dyn_atom_trace_backup/3`.

3.3 recwait

Within algo the two choices for representing algorithm state are mixed too much with the rest of the code. `recwait/0` is the switch between storage as recorded and storage as a dynamic clause. Sometimes code seems to be copy/pasted. But, it seems that backtracking in the recorded database and backtracking in the asserted database works differently, see `update_activity_times1/1`.

3.4 Following the progress of leadsto rules

We start with `setup_leadsto/6` where the arguments are almost identical to the values in the Leadsto specification.

setup_leadsto(*TStart, Vars, LitDisConj, AndLiterals, Delay,RId***)**
: where the arguments are almost identical to the values in the Leadsto specification. Then `init_interval_callbacks/9` transforms some constructs such as forall.

  After a number of steps involving normalizing conjunctions and disjunctions [7] and partial evaluation pruning out true and false results, `setup_lt/6` is called.

setup_lt(*Ante, Conse, Vars, Delay, Id, RId***)**
: The encoding of the antecedent is responsible for generating code. If a Term is a comparison operator, code is generated for that, if a term is an arithmetic expression, code is also generated.

  We pass on some (incomplete) data structures within `setup_lt/6`. `code_form/4` uses `ds_d(AnteResult, VarsIn, PVIn)` and `ds_d(AnteTail, VarsOut, PVOut)`. `AnteResult` is a difference list. Therefore often `AnteTail` is set to `[]`.

  `setup_lt/6` calls `setup_lt_normed/8`. The result is stored as `dyn_lt_rule(Id, AnteLits,ConseRId,PVOutC,Delay,RId)` but more important, `setup_lt_normed/8` is called, it leads to `setup_lt_wait_var/12`.

code_form(*+AnteConse, +PosNeg:[pos, neg], DIn, DOut***)**
: `code_form/4` is used for `Ante` and `Conse`.

  Each `AnteConseTerm` is translated as a list element in `AnteResult`. `L = ds_litd(Atom, PosNeg, PreOps, PostOps, PostConds)` where `Atom` can be true or any other value. It seems that its translated value is not tested in any way.

  Within `code_form/4`, `tr_arg_prolog1(Term, PVIn, Term1, Inst, DSTAIn, DSTAOut)` is used where

```
DSTAIn = ds_ta(VIn, PVIn, [], [], []),
...
DSTAOut = ds_ta(VOut, PVOut, PreOpsOut, PostOpsOut, PostCondsOut),
```

  Inst should result in `Inst == inst`.

tr_arg_prolog1(*Term, PVIn, Term1, Inst, DSTAIn, DSTAOut***)**
: `tr_arg_prolog1/6` translates leadsto variables into Prolog variables, their relationship is stored and retrieved in `PVIn/PVOut`, by `var_pl_to_var_list/6` and `var_pl_from_var_list/5`.

[7] TODO:Verify whether disjunction is allowed

The first encounter of a leadsto variable in a `code_form/4` has `Inst= next`, a later one gets `Inst = inst`. Inst values can be `inst`, `next`, `var`, `mixed`.

`tr_arg_prolog1/6` is also responsible for substituting *spec_constant*s. [8]

code_conse(*Conse, VOut, PVOut, Id, ConseRId, PVOutC*)
: translates `Conse` through `code_form/4`, but true `ConseLits` are removed. [9]

The consequent is encoded as `ds_cr(ConseLits, ds_ri(Id))`, but pxor consequents are treated differently. [10].

TODO: Looking at the code, it seems that we do not reorder the `AnteLiterals` depending on intermediate results.

4 Working backwards

Meaning, trying to reconstruct the algorithm from the start.

4.1 Leadsto times `e`, `f`, `g`, `h`

We limited `e, f, g, h`:`e,f, g, h >= 0` and if `h == 0` then `g` must be 0. But also, `e + f + g + h > 0`. [11]

`e, f, g, h`:

Once an antecedent holds for duration `g + T`, a delay is set between `e` and `f`, and the antecedent will hold during `h + T`. So, even if a rule has fired, we need to remember that it has fired and as long as the antecedent may continue to hold, the consequent will be propagated for a longer time.

4.2 Invariant

`HandledTime`: Everything that can be derived, has been derived for `T <= HandledTime`. CWA atom values do not have to be instantiated, probably will not be instantiated to false values.

4.3 Sketch of the algorithm

All rules that could still fire are inspected, their antecedent effect is exhaustively tested up to `HandledTime` at least.

After everything has fired, we inspect all waiting antecedents, and look at time their first result could come in. And the minimum value becomes the next `HandledTime`, unless this minimum value is not after `HandledTime` (could it be smaller?). It looks like the algorithm currently simply gives up if there is a rule that could fire at `HandledTime`.

It is probably important that together with setting `HandledTime`, every rule that has some continuation has its effect propagated till the new `HandledTime`.

This would make the invariant more precise: Every rule has its state updated in such a way that the antecedents have been checked up to the new `HandledTime`.

[8] TODO: Check whether Atoms could end up as Prolog variables, look at `code_atom/4` where `Inst` is ignored in the code. Can `Inst` be `var` or `mixed` there?

[9] (TODO: why not from Ante?).

[10] TODO

[11] Why those requirements? We probably do not want to reason without delay.

4.4 Rule States

Rules can contain variables, that is, antecedent literals can have variables. There can be more than one separate state per rule.

It could be that the first N literals of the antecedent with some specific instantiation are valid in some time range `T1 - T2`.

We will look strictly left to right.

But, the extending of the fired rules is done in reverse, why? Probably because we wish to extend the range as far as possible.

4.5 Garbage collection

Is complex and not documented. In the source some explanation is given, `olddocs/bugdev.txt` also contains some explanation.

4.6 Runtime Algorithm Predicates

setup_lt_normed(*AnteTODO, AnteHolds, TMin, THolds, ConseRId,PV, Delay, Removed***)**

We know AnteHolds holds in range *TMin*, *THolds*, we need to continue with *AnteTODO*. AnteHolds has all Prolog variables instantiated and excluded from `FVL` wait_var occurrences.

If *AnteTODO* becomes `[]`, `setup_lt_conse(AnteHolds, TMin, THolds, ConseRId, Delay, Removed)` is called. Otherwise, `setup_lt_notground/9` is called which calls `setup_lt_notground_fv/13` after setting up the free variable arguments.

setup_lt_conse(*AnteHolds, TMin, THolds, ConseRId, Delay, Removed***)**

We know The whole antecedent holds between *TMin* and *THolds*. If `THolds >= TMin + G`, we calculate `T3 is TMin + G + Delay` and `T4 is THolds + Delay + H` and then call `schedule_fire(ConseRId, T3, T4)` which sets `dyn_schedule_fire(ConseRId, T3, T4)`. [12] They are fired by the `repeat, set_state, handle_fired` sequence in `runspec_rest/0`. [13] After `schedule_fire/3` we do `setup_lt_wait_fired/6`. If `THolds < TMin + G` we call `setup_lt_wait_true/5`.

`setup_lt_wait_fired/6` stores a `wait_fired/5` fact. [14]

setup_lt_notground_fv(*+TStart, +FV, +FVL, +LitData, +ToDoAnte, +AnteHolds, +THolds, +ConseRId, +PV, +Delay, -*

AnteHolds holds for Time Interval between *TStart* and *THolds*. *ToDoAnte* is the conjunction that needs to hold. *LitData* is the Literal under investigation. *FV* are the free variables of the Literal and *FVL* is a list of instantiations that have been dealt with elsewhere.

Delay is `efgh(E,F,G,H)`, *ConseRId* is the consequent.

`Removed` indicates the source of the call. In case of `update_activity_time1(wait_var...)`, the `wait_var` term is Removed and the Removed is propagated along.

[12] I left out details dealing with pxor aspects. What is the reason for `schedule_fire/3`, why postpone?

[13] Why not fire immediately?

[14] It is confusing that two implementations of this waiting are present in the code, depending on the `recwait/0` switch.

PV is probably the characterization of the variables:`pv(Arga, Sorta,Kinda,Arg1a)`.

FV, *fv* are abreviations of Free Variables (Prolog variables).

`setup_lt_notground_fv/13` is called by `setup_lt_notground/9`, it by `setup_lt_normed/8`, it by `setup_lt/6`. `setup_lt/6` is called from `setup_nontrivial_leadsto/5`. `setup_lt_normed/8` is also called by `instantiate_op/16` (called in `add_default_cwa/17`) and `fail_filter_handleRR/16`, part of `filter_defaults_handle_others/14` in `setup_lt_notground_fv/13`.

We probably handle the rule in this call up to HandledTime.

Now, if at the call `THolds < HandledTime`, we start all over for this partially instantiated sequence of literals by calling `get_new_tholds/15`. Apparently the order of literals is reversed here. `get_new_tholds/15` is called here and by `get_new_tholds/15` itself.

If `THolds >= HandledTime` `setup_lt_notground_fv/13` analyses the current `LitData` first for all AtomTraces in `filter_defaults_handle_others/4` and all cwa matches in `add_default_cwa/17`, handles those separately by `check_fire_isolated/5` or `setup_lt_normed/8`. All those instantiations of *FV* handled separately are added to *FVL* and we finally call `setup_lt_notground1default/12` that calls `setup_lt_wait_var/12` with `Atom`, `FV`, `FVL` values implying that `FV` from `FVL` has been handled, but we need to check other instantiations.

get_new_tholds(*AnteHoldsTODO, AHDone, TStart, THoldsNew1, Tholds,FV, FVL, LitData, ToDoAnte, ConseRId,PV,Del*

Called by `setup_lt_notground_fv/13` and by `get_new_tholds/15` itself.

AnteHoldsTODO is an earlier instantiated sequence of literals that needs to be extended in range up to *THandled* (or further?). *AHDone* is the sequence that has been checked and hold between *TStart* and *THoldsNew1*. *Tholds* is the result. [15]

[16]

First, if `AnteHoldsTODO == []`, we continue with `setup_lt_notground_fv/13` with the increased time interval. Otherwise we follow *AnteHoldsTODO* Literals.

NEXT: What does `find_min_range_ground(Atom, PN, Tholds, O2)` do? Probably: Inspect Literal starting from Tholds.

setup_lt_wait_var(*FV, FVL, LitData, ToDoAnte, AnteHolds, TMin, THolds, ConseRId, PV, Delay, Id, IdTerm*)

Probably: We know *AnteHolds* is ok between *TMin* and *THolds*. *LitData* is the current literal that has been analyzed. FV are the free variables in the Literal, `FVL` is the list of instantiations of FV that have been dealt with, for which this setup is not responsible at all. Called by `setup_lt_notground1default/12` (same arguments) which is only called as last call in `setup_lt_notground_fv/13`.

TODO

- Really nail down the meaning of `wait_var`, also at what stage are what values for *TMin* and *THolds* set.

[15]TODO: Details of Tholds, is this a return parameter?

[16]When analysing the source:Be aware that in `get_new_tholds/15` the `Atom` in `[ds_lh(lit(Atom,PN),Id1,IdTerm1)|AnteHoldsTODO]` is ground, and has nothing to do with `FV` and `FVL`.

- Will `wait_vars` become invalidated? Inspect `get_new_tholds/15`.
- For a `wait_var` entry, can a result be found that extends *AnteHolds* starting at *TMin*? Probably: it could be that some rule has not fired yet, giving a new *LitData* instantiation. But please create an example.

5 So far

Try documenting the whole data structure that describes the state of each leadsto rule first. All invariants, the understanding of having every possible outcome of a leadsto rule represented.

At what stage the HandledTime invariant is. Understanding the get_new_tholds, the reverse is on purpose as that is part of the invariant, having a partial instantiation left to right.

After that, try understanding the cleanup efforts of `wait_var`.

6 "Syntax" of Leadsto Specifications

Copied from syntax.txt:

```
The leadsto input syntax is prolog syntax, but with the
following added/changed operator definitions.
(For input of leadsto specs in prolog, the : redefinition is
 awkward. I do a push/pop operator call for reading)

                        op(150, xfx, :),
                        op(700, xfy, <),
                        op(700, xfy, <=),
                        op(700, xfy, =<),
                        op(700, xfy, >),
                        op(700, xfy, >=)
                        ]).

Currently, only the top level terms are described. I am working
on syntax (+minimal explanation of semantics) of the top level
terms, but especially the sub terms.

sub terms:

% VAR:PLPCE:
    VAR, in principle, a prolog term, although Uppercase atoms
    are allowed. Quotes around atoms are allowed.
     TODO: are unquoted uppercase functors allowed?
     e.g. P(a) TODO: what are further restrictions and
     interpretations of PLPCE terms TODO: junk this stupid
     name "PLPCE".

% start_time(PLPCE)
```

```
% end_time(PLPCE)
% global_lambda(PLPCE)


TODO: why those qterms?
% qterm(cwa(X))        cwa_node
% qterm(external(X))   external_node
% qterm(X) ...         other_node

% display(_,_)
% display_number_range(_,_,_,_)

% periodic(Vars, Range, Period:PLPCE, Formula) is_list(Vars)
                 * handle_interval(Vars, Range, Formula, Root, Son, Extra)
% periodic(ST, ET, Period:PLPCE, Formula)
                 * handle_interval([], range(ST, ET), Formula, Root,Son,Extra),
% periodic(Vars, ST, ET, Period:PLPCE, Formula)
                 * handle_interval(Vars, range(ST, ET),Formula,Root,Son,Extra)
% interval(Vars, ST, ET, Formula)
                 * handle_interval(Vars, Range, Formula, Root, _Son, Extra)
% interval(Vars, ST, ET, Formula)
                 * handle_interval(Vars, range(ST, ET),Formula,Root,_Son,Extra)
% interval(ST, ET, Formula)
                 * handle_interval([], range(ST, ET), Formula,Root,_Son,Extra)

% leadsto(AnteFormula, ConseFormula, Delay)
        * handle_leadsto1(Root, AnteFormula, ConseFormula, Delay, Extra, _Son)
% leadsto(Vars, AnteFormula, ConseFormula, Delay)
        * handle_leadsto1(Root, AnteFormula, ConseFormula, Delay, Extra, Son)
% specification(_)
                   * IGNORED
% content(C)
                * TODO? assertz(dyn_content(C))
% denotes(Header, Formula)
                  * term_to_formula_node(Formula, FormulaNode, Extra),
                new(PN, property_def_node(@off)),
                send(PN, fill_header, Header),
                send(PN, son, FormulaNode),
                send(Root, son, PN)
% (sort_element(SortName:PLPCE, Term):- member(Term2, List) with Term==Term2
                * test_sort_def(SortName, List, Extra),
        ensure_sort_son(Root, SortName1, SNode),
        add_sort_contents(SNode, List).
% constant(Name, Value)
  * check_constant(Name, Value),
        send(Root, son, new(N, constant_def_node)),
        send(N, fill_header, Name),
```

```
        send(N, fill_value, Value)
% sortdef(SortName:PLPCE, Objs)
  * test_sort_def(SortName, Objs, Extra),
        send(Root, son, new(SN, sort_node)),
        send(SN, change_gui_prop, sort_name, SortName1),
        add_sort_contents(SN, Objs).
% cwa(PLPCE)
% model(PLPCE)
% [specification_element]
  * generic node
```