The PTC Solver User Manual Version 1.5.3

Dr. Christophe Meudec
Ireland

February 22, 2005

Abstract

The PTC Solver is a constraints solver over path traversal conditions in Pascal, C, Ada, Java Bytecode like programming languages without pointers. It can be used for a variety of purposes in the area of software program analysis.

Keywords: constraint logic programming, program analysis

1 Introduction

The PTC Solver described here is primarily intended for program analysis purposes. It roughly covers types and expressions that can be encountered in the Pascal programming language (except pointers). Constraints over integers, rationals, enumerations, boolean organised into records and arrays can be submitted. The solver adds constraints to its own internal constraint store for evaluation. A satisfiable set on constraints can be sampled and the solver generates actual values which reduces the system of constraints to true.

To use the PTC Solver to its full potential a knowledge of the Prolog programming language is necessary.

2 Referencing

The main reference for the PTC solver is Christophe Meudec, ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution in Software Testing, Verification and Reliability Journal June 2001, Vol 11, pp. 81-96.

3 Installation

The PTC Solver uses the ECLiPSe contraint logic programming environment. Therefore it will not work unless you install the latest version of ECLiPSe on your machine. ECLiPSe is free for non-commercial purposes and can be downloaded once the license agreement has been signed from `http://www.icparc.ic.ac.uk/eclipse/`. ECLiPSe is available on the Windows and Unix platforms.

The Zip file containing the PTC Solver you downloaded should be extracted in your ECLiPSe intallation directory. The PTC Solver will install itself in the `lib_public` subdirectory of ECLiPSe.

The way to load the solver depends on your programming environment (Prolog, C/C++, Tcl/Tk or Visual Basic) as explained in the relevant sections below.

The PTC Solver has been primarily developed for the Windows family of operating systems. It should be directly portable to Unix platforms using the appropriate ECLiPSe Unix version. However the embedding and interfacing issues discussed in this manual are mostly only applicable to Windows.

4 ATGen's Solver Basics

Here we describe briefly the fundamental concepts of the solver by examining some of its most simple interface calls.

You can declare real, integer, enumeration, record and array variables and impose constraints on them. The solver handles linear as well as non-linear constraints. On submitting a constraint the solver adds it to its existing store of constraints and can then:

- fail, the system of constraints was unsatisfiable, the system of constraint remains as it was before the addition of the latest constraint (the solver backtracks automatically)
- succeeds, the system of constraints may be satisfiable (if non-linear constraints are present the solver may fail to detect their unsatisfiability at this stage)

A satisfiable system of constraints can be sampled to generate actual values which reduces the set of constraints to true.

5 Limitations and Issues

We list here intrinsic limitations of the PTC solver as well as issues users should be aware of.

5.1 Integer Overflowing

The PTC solver is dependent on the ECLiPSe system for the manipulation constraints over integers. Currently, ECLiPSe is prone to integer overflow without warning. Thus spurious results may be obtained without warning. All integers are coded on 32 bits.

To avoid overflow as much as possible the maximum range for integer in the PTC solver is relatively small.

Note that this limitation also applies to the operands of the bitwise and shifting constraints. Thus only 16 bits encoded decimals can be handled properly. Constraints on 32 bits and 64 bits encoded decimals actually use only 16 bits in the solver. This limitation will often result in an unsound solver for these encodings.

5.2 Numeric Precision Issues

Internally the PTC solver represents real numbers involved in linear constraints using infinite precision reals. Precision is reversed to double floating point precision whenever non-linear constraints are applied.

Rational numbers are returned by the PTC solver. To avoid further loss of precision it is recommended that you use rationals in your application. The subsection *Useful for Embedding* describes predicate useful for the manipulation of rational numbers.

The ptc_solver__rational_to_decimal predicate is also provided to convert a rational number into a double precision decimal point number.

5.3 Logical Connectors Priorities

In the PTC solver the priority of the logical connectors is different than what is commonly expected. See the *Constraint Syntax* subsection for details.

5.4 The 'or' Constraint

The 'or' constraint has two modes of behaviour, which can be set during initialisation using the ptc_solver__set_flag(or_constraint_behaviour, Value) predicate where Value is one of the following ('pure' is the default value):

- 'pure', here the 'or' constraint behaves as a true constraint. In other words as much as possible is deduced from the constraint and it remains latent in the system until more information can be deduced automatically.
- 'choice', here the 'or' constraint is broken down and its components submitted to the system one by one during backtracking. The extra induced backtracking on failure must be tackled appropriately at a higher level. Thus on backtracking the 'or' constraint returns different ways in which it can be satisfied. Formally the constraint 'A or B' can be satisfied by 'A and not(B)', 'A and B' and 'not(A) and B'.

5.5 Randomness

As much as possible the PTC solver behaves non-deterministically. Do not expect to get twice the same result on the same example on different runs. Labeling is non-deterministic, even the 'and' constraint will not behave always the same way as to the order of the sub-constraints that are submitted to the system.

6 Interfacing

The PTC Solver is written in Prolog using the ECLiPSe programming environment. ECLiPSe has itself interfacing capabilities to C/C++ and Visual Basic. Therefore you can implement your application:

- in Prolog using ECLiPSe
- in C/C++
- in Visual Basic, Tcl/Tk or Java

The solver provides a number of Prolog predicates which defines its interface. These are detailed in the final section of this manual.

6.1 Using the solver via ECLiPSe Prolog

This is probably the most straightforward way of using the solver as the interfacing is then seamless.
As an example consider the following program compile by Eclipse.

```
:- import ptc_solver.                              %the solver is compiled and
imported

example(A, B) :-
        ptc_solver__clean_up,                      %to start in a clean
environment
        ptc_solver__default_declarations,          %solver initialisations
        ptc_solver__variable([A, B], integer),     %A and B are declared as
integers
        ptc_solver__sdl(A>45 and B-5=A*A),         %constraints are imposed
        ptc_solver__label_integers([A,B]).         %a unique random solution is
generated
```

Eclipse answer:

```
?- example(A, B).
A = 77
B = 5934
More (0.00s cpu)
```

All labeling predicates generate alternative values on backtracking.

6.2 Using the C/C++ Interface

Here we can write a C/C++ program and use the standard interfacing capabilities of ECLiPSe. In order to use this facility it is strongly recommended that you familiarise yourself with ECLiPSe's Embedding and Interfacing Manual.

The examples given here are available in the `ptc_embed_solver.cpp` and `ptc_embed_sessions.cpp` files.

What follows is only applicable to the Windows operating system. Embedding of the PTC Solver on a Unix platform has not been attempted. The reader is referred to the ECLiPSe's Embedding and Interfacing Manual for embedding in a C/C++ application in a Unix environment.

For Windows, and despite the official ECLiPSe documentation, it has proved impossible to embedded the PTC Solver in a C/C++ application other than by using the Visual C++ compiler. In particular the reader is to be warned that the embedding capabilities of ECLiPSe seem incompatible with the Borland family of compilers.

The embedding examples given here are therefore to be compiled with a Visual C++ compiler. If a different compiler is preferred for the development of the application then it is suggested that a simple Visual C++ interface be created and that a Windows DLL be generated. The application can then interact with the solver via this DLL compiled using Visual C++.

The reader must refer to ECLiPSe's Embedding and Interfacing Manual to follow the examples given below. Other modes of interactions than given in the examples are feasible through the implementation of auxilliary C/C++ functions.

6.2.1 Simple Example

To facilitate the submission of constraints to the PTC Solver from C/C++ a special predicate of the PTC Solver interface is available: `ptc_solver__submit_string/1`. It allows the posting of constraints and declarations in a string format. Variables must be Prolog like (for example they must start with a uppercase letter). This predicate is easily called using the auxillary C function: `submit_string` whose code is given below.

Note that the use of strings for constraints submission is not absolutely necessary: developpers can declare Prolog like variables in their C/C++ code and construct goals to be posted to the PTC Solver using ECLiPSe's interfacing `term` function. Refer to ECLiPSe's Embedding and Interfacing Manual.

```
// adds a constraint in string format to the current store of
//  constraints and returns EC_succeed on success and EC_fail otherwise
int submit_string(char *s)
{
post_goal(term(EC_functor("ptc_solver__submit_string", 1), s));
return EC_resume();
}//submit_string
```

A basic embedding example is then:

```
#include <eclipseclass.h> //necessary for embedding ECLiPSe into C/C++
#include <stdio.h>
void main()
{
ec_set_option_ptr(EC_OPTION_DEFAULT_MODULE, "ptc_solver"); // initialisation of
ECLiPSe
ec_init();

post_goal(term(EC_functor("lib", 1), "ptc_solver")); // loading of the solver
printf("ECLiPSe and the PTC Solver are being loaded ...\n");
if (EC_resume()==EC_succeed) printf("The PTC Solver has been loaded\n");
else printf("Error: The PTC Solver did not load properly\n");

submit_string("ptc_solver__clean_up,
               ptc_solver__default_declarations"); // initialisations of the
solver
printf("The PTC Solver is initialised\n");

submit_string("ptc_solver__variable([A, B], integer),
submit_string("ptc_solver__sdl(A>45 and B-5=A*A)");
submit_string("ptc_solver__label_integers([A,B])");

//auxillary C function to retrieve and display the values of all the variables
display_all_vars();

ec_cleanup(); //Unloading ECLiPSe and tidying up

printf("ECLiPSe and the PTC Solver have been unloaded\n\n");
}
```

The auxillary `display_all_vars()` C function, retrieves and displays the value of all submitted variables. It should be modified to suit your application.

6.2.2 Backtracking

Even when embedding the PTC Solver into a C/C++ application we can make the most of its backtracking capabilities. The following example illustrate this.

Imagine that we want to generate tests to cover in its entirety the control flow graph of a simple program code fragment. Instead of submitting constraints individually we can undo the last constraint submission using the backtracking facility of the PTC Solver, which in this regard behaves in an ordinary Prolog program manner.

It is however slightly more difficult than using Prolog directly. We use tell-tale variables to indicate whether a submitted constraint has failed or succeeded. If a constraint is submitted that fails then the solver backtracks automatically. The auxillary `check_success` C function has been especially written to check the values of those tell-tale variables.

```
//Demonstration of search in a control flow graph
//Corresponds roughly to the code:
// if X>2 or Y>2 then blah; //Condition 1
//   else blah;
// if X<2 then blah;        //Condition 2
//   else blah;
// where blah does not change the values of X and Y
// we check the paths in the following order:
//       1 true, 2 true
//       1 true, 2 false
//       1 false, 2 true
//       1 false, 2 false
void session3()
{
fprintf(log, "___STARTING SESSION 3___\n");

  //X and Y are declared as integers
submit_string("ptc_solver__variable([X, Y], integer)");

  //The general format for constraints submission that can
  // backtrack is "Si = 1, Constraint ; Si = 0"
submit_string("S1 = 1, ptc_solver__sdl(X>2 or Y>2) ; S1 = 0");
  //The values of the variables Si indicate if the constraint has
  // been successful (1) or not (0)
submit_string("S2 = 1, ptc_solver__sdl(X<2) ; S2 = 0");
  //Check_success should return 1
fprintf(log, "S2 is %i\n", check_success("S2"));
  //Check_success will return -2 as S10 does not exist
fprintf(log, "S10 is %i\n", check_success("S10"));
  //A sample is generated for X and Y
submit_string("ptc_solver__label_integers([X, Y]), !");
  //The values of current variables are displayed. Should satisfy: X<2 and Y>2
display_all_vars();

  //We backtrack one level on purpose: the constraint X<2 is undone (S2 will now
be 0)
submit_string("fail");
  //Check_success should return 0 for S2
fprintf(log, "S2 is %i\n", check_success("S2"));
submit_string("S3 = 1, ptc_solver__sdl(not(X<2)) ; S3 = 0");
submit_string("ptc_solver__label_integers([X, Y]), !");
display_all_vars(); //Should satisfy: X>2 or (X=2 and Y>2)

submit_string("fail"); //We backtrack on purpose S3 = 0
submit_string("fail"); //We backtrack on purpose S1 = 0
  //S2 has fallen out of scope: check_success returns -2
fprintf(log, "S2 is %i\n", check_success("S2"));
submit_string("S4 = 1, ptc_solver__sdl(not(X>2 or Y>2)) ; S4 = 0");
submit_string("S5 = 1, ptc_solver__sdl(X<2) ; S5 = 0");
submit_string("ptc_solver__label_integers([X, Y]), !");
display_all_vars(); //Should satisfy: X<2 and Y<=2

submit_string("fail");  //S5 = 0
submit_string("S6 = 1, ptc_solver__sdl(not(X<2)); S6 = 0");
submit_string("ptc_solver__label_integers([X, Y]), !");
display_all_vars(); //Should satisfy: X=2 and Y<=2

fprintf(log, "___END SESSION 3___\n");
}//session3
```

6.2.3 Controlling Backtracking on Failure

Of course sometimes a submitted constraints will fail. Below is such an example and illustrates how to control the backtracking in this circumstance.

```
//Demonstration of search in a control flow graph with failure of
// one of the path
//Corresponds roughly to the code:
// if X<=2 and Y<=2 then blah;
//                  else blah;
// if X<=2 then blah;
//         else blah;
// where blah does not change the values of X and Y
void session4()
{
fprintf(log, "___STARTING SESSION 4___\n");
submit_string("ptc_solver__variable([X, Y], integer)");

submit_string("S1 = 1, ptc_solver__sdl(X<=2 and Y<=2) ; S1 = 0");
submit_string("S2 = 1, ptc_solver__sdl(X<=2) ; S2 = 0");

submit_string("ptc_solver__label_integers([X, Y]), !");
display_all_vars();     //Should satisfy : X<=2 and Y<=2

  //We backtrack one level on purpose: the constraint X<=2 is undone (S2 will now
be 0)
submit_string("fail");
  //This should fail as (X<=2 and Y<=2) and X>2 is a contradiction;
  // X>2 could not be added indicated by S3 = 0
submit_string("S3 = 1, ptc_solver__sdl(not(X<=2)) ; S3 = 0");
submit_string("ptc_solver__label_integers([X, Y]), !");
display_all_vars(); //Should satisfy: X<=2 and Y<=2

submit_string("fail");

submit_string("S4 = 1, ptc_solver__sdl(not (X<=2 and Y<=2)) ; S4 = 0");
submit_string("S5 = 1, ptc_solver__sdl(X<=2) ; S5 = 0");
submit_string("ptc_solver__label_integers([X, Y]), !");
display_all_vars(); //Should satisfy: X<=2 and Y>2

submit_string("fail");
submit_string("S6 = 1, ptc_solver__sdl(not(X<=2)); S6 = 0");
submit_string("ptc_solver__label_integers([X, Y]), !");
display_all_vars(); //Should satisfy: X>2

fprintf(log, "___END SESSION 4___\n");
}//session4
```

6.2.4 Conclusion

The interactions described above are difficult to trace and therefore debug.

If the behaviour is not as expected, an attentive manual trace of the posted goals and an examination of the tell-tale variables should be undertaken. For debugging interfacing using Prolog is probably the most effective solution.

6.3 Using the Visual Basic, Tcl/Tk or Java interface

The reader is entirely referred to the ECLiPSe embedding manual for embedding the PTC Solver into those environments. This has not been attempted by the author of the PTC Solver.

7 The Interface

Here we describe in full the Prolog predicate that can be called once the solver is loaded.

7.1 Terminology

Type_mark designate the name of a type definition (e.g. speed, money), it should start with a lower case letter.

Min, Max expressions designating the minimum (resp. the maximum) in a range. (e.g. X + 1, 100). By default the integer range is -65535..65535 and the real range $-2^{40}/3.. + 2^{40}/3$. Reals are internally implemented using infinite precision rational numbers.

Literal_list is an ordered Prolog list of literals for enumeration types (e.g. [mon, tue, wed, thu, fri, sat, sun])

Field_list is an ordered Prolog list of elements of the form (Field_name_list, Type_mark) and Field_name_list is a list of field_names (e.g. [([father, mother], parent_type), ([age], integer)]). field_names should start with a lower case letter.

Index_list is an ordered Prolog list of integer or enumeration Type_marks or Subtype_marks. The range of each element of the list designate the range of the corresponding index in the array (e.g. [name_t] : implies one dimentional array index from ptc_solver__first(name_t) to ptc_solver__last(name_t). [name_t, name_t] implies a two dimentional array.)

Component_type_mark synonym of Type_mark

Subtype_type_mark synonym of Type_mark

Identifier_list a list of identifiers. Identifiers must respect the following BNF format (taken from the Eclipse user manual): (UC | UL) ALP* where UC denotes all upper case letters, UL is the underline charater, and ALP is (UC | UL | LC | N) where LC denotes all lower case letters and N is any digits. (e.g. _ry784A, A__676, Money)

7.2 Initialisations

`ptc_solver__clean_up`
Erases all previous declarations
e.g. `ptc_solver__clean_up.`

`ptc_solver__version(Ver)`
Returns as a string the version of the PTC solver in use
e.g. `ptc_solver__version(Ver) returns Ver = "1.2.3"`

`ptc_solver__default_declarations`
Declares the following default types: integer : $-65535..+65535$ float : $-2^{40}/3..+2^{40}/3$ and Boolean
e.g. `ptc_solver__default_declarations.`

`ptc_solver__set_flag(Flag, Value)`
Allows the PTC solver to be customised. The following flags are provided:

- or_constraint_behaviour, where Value must be choice or pure, the default is pure. See the subsection on the 'or' constraint in the Limitations and Issues section. This flag should be set once and only once, after the call to ptc_solver__clean_up but prior to anything else.
- enumeration_start, where Value is an integer, the default is 1. This flag indicates the starting point for the numbering of enumeration literals. It can be set as many times as wanted prior to the definition of an enumerated type.
- float_to_int_convention, where Value must be truncate or nearest, the default is truncate. This flag customise the behaviour of the explicit type conversion from real to integer. The truncate value indicates that C convention is used e.g. 3.6 becomes 3. The nearest value indicates that the Ada convention is used e.g. 3.6 becomes 4. This flag should be set once and only once, after the call to ptc_solver__clean_up but prior to anything else.

e.g. `ptc_solver__set_flag(or_constraint_behaviour, pure)`

7.3 Declarations

7.3.1 Types

`ptc_solver__type(Type_mark, real)`
Declares a real type
e.g. `ptc_solver__type(balance, real)`

`ptc_solver__type(Type_mark, real, range_bounds(Min, Max))`
declares a real type bounded between Min and Max
e.g. `ptc_solver__type(speed, real, range_bounds(0.0, 165.0))`

`ptc_solver__type(Type_mark, integer, range_bounds(Min, Max))`
declares an integer type bounded between Min and Max
e.g. `ptc_solver__type(year_t, integer, range_bounds(1900, 3000))`

`ptc_solver__type(Type_mark, enumeration, Literal_list)`
declares an enumeration type
e.g. `ptc_solver__type(name_t, enumeration, [mon, tue, wed, thu, fri, sat, sun]))`

`ptc_solver__type(Type_mark, record, Field_list)`
declares a record type
e.g. `ptc_solver__type(date_t, record, [([name], name_t), ([day], day_t), ([month], month_t), ([year], year_t)])`

`ptc_solver__type(Type_mark, array, Index_list, Component_type_mark)`
declares an array type of dimensions determined by Index_list and whose elements are of type Component_type_mark
e.g. `ptc_solver__type(week_t, array, [name_t], date_t)`

7.3.2 Subtypes

`ptc_solver__subtype(Subtype_mark, Type_mark)`
Declares a subtype for the type_mark
e.g. `ptc_solver__subtype(argent, money)`

`ptc_solver__subtype(Subtype_mark, Type_mark, range_bounds(Min, Max))`
Declares a subtype of the type Type_mark with new range Min, Max

e.g. `ptc_solver__subtype(week_day, name_t, range_bounds(mon, fri))`

`ptc_solver__subtype(Subtype_mark, Type_mark, range(_))`
TODO
e.g.

`ptc_solver__subtype(Subtype_mark, Type_mark, range(_, _))`
TODO
e.g.

7.3.3 Variables

`ptc_solver__variable(Identifier_list, Type_mark)`
Declares all the variables in Identifier_list to be of type Type_mark
e.g. `ptc_solver__variable([NEXT_DATE, DATE], date_t)`

7.4 Ranges

`ptc_solver__integer_range(Identifier, Min, Max)`
Returns the minimum and maximum of the range of an Integer variable. Based on the linear constraints imposed on the variable.
e.g. `ptc_solver__integer_range(Day, mon, fri)`

`ptc_solver__real_min(Identifier, Inf, Taken)`
Returns the minimum, Inf, of the range of a Real variable, Identifier. Taken is 'taken' if the lower bound is inclusive and not_taken if it is exclusive
e.g. `ptc_solver__real_min(Speed, 0.0, taken) succeeds`

`ptc_solver__real_max(Identifier, Sup, Taken)`
Returns the maximum, Sup, of the range of a Real variable, Identifier. Taken is 'taken' if the upper bound is inclusive and not_taken if it is exclusive
e.g. `ptc_solver__real_max(Speed, 165.0, taken) succeeds`

`ptc_solver__first(Type_mark, Min)`
Returns Min the smallest value defined by the type, Type_mark
e.g. `ptc_solver__first(day, mon)`

`ptc_solver__last(Type_mark, Max)`
Returns Min the largest value defined by the type, Type_mark
e.g. `ptc_solver__last(day, sun)`

7.5 Constraint Submission

`ptc_solver__sdl(Constraint)`
Adds Constraint to the current constraint store. Fails if the constraint store becomes unsatisfiable. A constraint is a Boolean expression which must respect the syntax defined below.
e.g. `ptc_solver__sdl(X > Y and X/2 = 5)`

`ptc_solver__arithmetic(Expression, Result, Type)`
Expression is an arithmetic expression (without Boolean operators and relational operators), Result is a variable representing the operation denoted by Arithmetic, Type is r for real, i for integer, e for enumeration, record for record, array for array, boolean for Boolean
e.g. `ptc_solver__arithmetic(X*Y, Z, Type)` Usage is not recommended in normal circumstances, use ptc_solver__sdl instead

`ptc_solver__relation(Relation, Left, Right)`

Relation is a relational operator, Left is its left operand and Right is its right operand e.g. `ptc_solver__arithmetic(>, X*Y, Z)` Usage is not recommended in normal circumstances, use ptc_solver__sdl instead

7.6 Constraint Syntax

Expressions and Contraints must respect the following syntax.

Note that the priority of the logical connectors is different than what is commonly expected (it is in fact Ada-like here). In particular the operators **and**, **or** and **xor** have the same priority. Thus Boolean expressions must be written with brackets as in A **and** (B **or** C).

Most of below is self explanatory for users with knowledge of programming.

Mixed type arithmetic (i.e. expression involing reals and integers) is handled in C fashion. So an integer expression involving mixed arithmetic will evaluate to an expression of type real.

To avoid mixed constraints the `conversion` constraint can be used to convert reals into integers (the value of the flag float_to_int_convention controls the behaviour of this transformation), or to convert integers into reals. For example `ptc_solver__sdl(B = 2.6)` where `B` is an integer will fail but `ptc_solver__sdl(B = conversion(integer, 2.6))` results in `B` becoming `2` if the float_to_int_convention flag is set to its default value : truncate.

The constraint `eq_cast` allows the modelling of C 'assignments'. It is similar to the `=` constraint except whenever in C an integer is assigned a float. For example `ptc_solver__sdl(eq_cast(B, 2.6))` where `B` is an integer results in in `B` becoming `2` if the float_to_int_convention flag is set to its default value : truncate. In effect eq_cast automatically performs the implicit type conversion from float to integer when necessary as would be performed during the assignement of a float expression to an integer variable in C. `eq_cast(X, Y)` is to be interpreted as `X` is equal to `Y` with `Y` being implicitly converted to the type of `X`. Thus `eq_cast(X, Y)` has not the same meaning as `eq_cast(Y, X)`. In C, unless all implicit casting can be eliminated using the `conversion` constraint, assignements (`=`) should be modelled using the `eq_cast` constraint, while equality (`==`) should be modelled using the `=` constraint. Languages such as Ada, and Java Bytecode are more reasonable and do not allow implicit casting: there is no need to use `eq_cast`.

The `round` constraint rounds a real to its nearest integer so that `ptc_solver__sdl(B = round(2.6))` results in `B` becoming `3`.

The **reif** constraint takes in a Boolean expression and a integer tell-tale variable betwen 0 and 1. The tell-tale variable can be used to add the constraint itself to the current system of constraints (if set to 1) or its negation (if set to 0). If the constraint cannot be negated the tell-tale variable will be set to 1 automatically. If the constraint itself is not satisfiable the tell-tale variable will be set to 0.

Informally, the **cmp** constraint takes in two float expressions and an integer expression. The constraint behaves as follows: if the first argument is larger than the second then the third is 1, if the first argument is smaller than the second then the third is -1, if the first argument is equal to the second then the third is 0.

The **element** manipulator returns the element of an array as indexed by the expression. The **field** manipulator returns the element of a record as indicated by the field identifier. The **up_arr** update, returns a new array with the element at the position indexed updated to the expression given. The **up_rec** update, returns a new record with the indicated field updated by the expression given.

contraint : exp | **reif(** exp **,** *variable*_identifier **)** | **eq_cast(** exp **,** exp **)** | **cmp(** sum **,** sum **,** sum **)**
exp : negation { (**and** | **and_then** | **or** | **or_else** | **xor**) negation }
negation : [**not**] relation
relation : sum [(`=` | `<>` | `<` | `>` | `<=` | `>=`) sum]
sum : [**-**] term {(**+** | **-**) term }
term : factor {(***** | **/** | **mod** | **rem**) factor }
factor : primary | primary ****** primary
primary : *variable*_identifier | literal | natural | float | rational
| *constant*_identifier | function_designator | (exp)

natural : digit { digit }

float : natural **.** natural [(e | E) natural]
rational : natural **_** natural

```
function_designator : attribute | constructor | manipulator | arithmetic_function
attribute : first( type_mark ) | last( type_mark )
          | succ( exp ) | pred( exp )
          | pos( type_mark, exp ) | val( type_mark, exp )
arithmetic_function : conversion( type_mark , sum ) | round( sum ) | abs( sum )
                    | bw_not( exp , encoding_length , encoding_scheme )
                    | bw_and( exp , exp , encoding_length , encoding_scheme )
                    | bw_or( exp , exp , encoding_length , encoding_scheme )
                    | bw_xor( exp , exp , encoding_length , encoding_scheme )
                    | left_shift( exp , exp_amount , encoding_length , encoding_scheme )
                    | right_shift( exp , exp_amount , encoding_length , encoding_scheme )
encoding_length : 8 | 16 | 32 | 64
encoding_scheme : signed | unsigned
constructor : mk_record | mk_array
mk_record : agg(record_type_identifier , [argument_list])
argument_list : positional_argument_list | named_argument_list
positional_argument_list : exp { , exp }
named_argument_list : association { , association }
association : ([field_identifier], exp)

mk_array : agg(array_type_identifier,[element _list])
element_list : positional_element_list | named_element_list
positional_element_list : exp { , exp }
                        | exp { , exp } , ([others], exp )
named_element_list : ([others], exp )
                   | assign_list
                   | assign_list , ([others], exp )
assign_list : assign {, assign}
assign : (index_list, exp)
index_list : [index {, index}]
index : exp | range_bounds(exp, exp) | range(type_mark)

manipulator : element | field | update
element : element(array_expression, simple_index)
simple_index : [exp { , exp } ]
field : field(record_expression, field_identifier)

update : array_update | record_update
array_update : up_arr(array_expression, simple_index, exp)
record_update : up_rec(record_expression, field_identifier, exp)
```

7.7 Labeling

Labeling strategies can make a dramatic difference in the time needed to label the variables according to the current system of constraints. Labeling strategies in the PTC solver need improvement (but nothing prevents the user to define their own labeling predicates instead of using the default ones detailed below). Here are a few recommendations:

- It is essential that all variables of the same type be submitted all at once to the appropriate labeling predicate. This allows the predicate to select the most promising variables first.
- The labeling predicate implements some randomness, therefore different runs can yield vastly different runtimes. Sometime labeling can get lucky are return quickly. Sometime labeling will be bogged down.
- Labeling of integer variable is exhaustive i.e. all integer values in the constrained range will be tried. If there is no solution to the current store of constraint (e.g. because of non-linear constraints) labeling will typically take a long time and eventually fail. If the integer labeling fails it is because there are no solution.

- Labeling of real variables in non exhaustive (it couldn't be otherwise). If real labeling fails it does not follow that the system of constraint is unsatisfiable.
- It is impossible to decide without knowing the details of the current constraints the order in which labeling should be performed. Sometime labeling integers before reals will be optimal, at other time the inverse will be true.
- If your labeling gets bogged down there are a number of strategies you can use short of writing your own labeling predicates. First a timer can be set to stop a given labeling predicate and then start again. Another solution worth a try is to change the labeling order (e.g. try labeling reals first and then integers, or vice versa). A combination of both tips can also be implemented.

`ptc_solver__label_integers(Identifier_list)`
: Instantiate the integer variables in Identifier_list to values consistent with the current constraint store. Fails if no solution exist. An exhaustive search is performed so that on failure it can be concluded that the store of constraint was inconsistent.
e.g. `ptc_solver__label_integers([X, Y, Z])`

`ptc_solver__label_reals(Identifier_list)`
: Instantiate the float variables in Identifier_list to values consistent with the current constraint store. Can fail if no solution is found. The search is not exhaustive so that on failure it cannot be concluded that the store of constraint was inconsistent.
e.g. `ptc_solver__label_reals([Speed, Altitude])`

`ptc_solver__label_enums(Identifier_list)`
: Instantiate the enumeration variables in Identifier_list to values consistent with the current constraint store. Fails if no solution exist. An exhaustive search is performed so that on failure it can be concluded that the store of constraint was inconsistent. Boolean should be labelled using this predicate since Booleans are treated as enumeration variables.
e.g. `ptc_solver__label_enums([Next_day, Today])`

`ptc_solver__sample_enum(Identifier)`
: Instantiates Identitifer which is an enumeration variable to a sample value.
e.g. `ptc_solver__sample_enum(Day)` Usage is not recommended in normal circumstances, use ptc_solver__label_enums([Indentifier]) instead

7.8 Useful for Embedding

`ptc_solver__submit_string(String)`
: Allows the submission of goals using strings. See the embedding via C/C++ section of the user manual
e.g. `ptc_solver__submit_string("ptc_solver__sdl(X>Y*Y+Y and Y<5)")`

`ptc_solver__get_single_variable(String, Variable)`
: Retrieves the actual variable of a variable submitted in a string using ptc_solver__submit_string(String)
e.g. `ptc_solver__get_single_variable("S1", Value)` Returns -2 if the Identifier is out of scope

`ptc_solver__get_all_variables(Name_variable_list)`
: Retrieves the list of Name, Variable couples for all variable submitted via a string using ptc_solver__submit_string(String). See the auxillary C function display_all_vars
e.g. `ptc_solver__get_all_variables(Var_list)`

`ptc_solver__is_rational(Rat)`
: Succeeds if Rat is a rational number (i.e. represents a floating point number).
e.g. `ptc_solver__is_rational(Speed)` See the ptc_solver__rational_to_decimal predicate to obtain a decimal representation of rational numbers useful for embedding

`ptc_solver__rational_to_decimal(Rat, Dec)`

Dec is the double precision decimal representation of Rat. Note that there is an inevitable loss of precision for non decimal numbers.
e.g. `ptc_solver__rational_to_decimal(2/3, 0.66666666666666663)` You are encouraged to carry on working with infinite precision rationals to avoid precision problems.

`ptc_solver__numerator(Rat, Num)`
Num is the numerator of the rational Rat
e.g. `ptc_solver__numerator(Rat, 2)`

`ptc_solver__denominator(Rat, Den)`
Den is the denominator of the rational Rat
e.g. `ptc_solver__numerator(Rat, 3)`

7.9 Micellanous

7.9.1 Type testing

`ptc_solver__is_enum(Identifier)`
Succeeds if Identifier is an enumeration variable, fails otherwise
e.g. `ptc_solver__is_enum(Today)`

`ptc_solver__is_record(Identifier)`
Succeeds if Identifier is a record variable, fails otherwise
e.g. `ptc_solver__is_record(Date)`

`ptc_solver__is_array(Identifier)`
Succeeds if Identifier is an array variable, fails otherwise
e.g. `ptc_solver__is_array(Marks)`

`ptc_solver__is_integer(Identifier)`
Succeeds if Identifier is an integer variable, fails otherwise
e.g. `ptc_solver__is_integer(RPM)`

`ptc_solver__is_real(Identifier)`
Succeeds if Identifier is a float variable, fails otherwise. Note that float variables are represnted as rationals
e.g. `ptc_solver__is_real(Speed)`

7.9.2 Internal Declaration Manipulation

`ptc_solver__get_frame(Type_mark, Basetype_mark, Identifier)`
Given a Type_mark, this predicate will return its basetype and a variable Identifier of type Type_mark. Basetype is one of the following: real, integer, base_enumeration, record, array(Component_type_mark) or Type_mark which then denotes that Type_mark is a subtype of Basetype_mark. Note that this predicate is very slow whenever a lot of types have been declared.
e.g. `ptc_solver__get_frame(money, B, Frame)`

7.9.3 Retrieving components

`ptc_solver__get_array_index_elements(Identitifer, Index_element_list)`
Given an array variable Identifier, returns its Index_element_list. Index_element_list for an array of dimension 1..3, 1..2 is of the form: [([1,1], el1.1), ([1,2], el1.2), ([2,1], el2.1), ([2,2], el2.2), ([3,1], el3.1), ([3,2], el3.2)] where el*.* denotes the actual variables in the array
e.g. `ptc_solver__get_array_index_elements(Marks, List)`

`ptc_solver__get_record_field_values(Identifier, Field_value_list)`

Given a record variable Identifier, returns its Field_value_list. Field_value_list is of the form [(field1, value1), ..., (fieldn, valuen)]
e.g. `ptc_solver__get_record_field_values(Date, Fields)`

7.9.4 Enumeration Variables Manipulations

`ptc_solver__enum_get_literal(Type_mark, Value, Literal)`
Returns the Literal of a Type_mark corresponding to Value
e.g. `ptc_solver__enum_get_literal(week_day, 2, tue) succeeds`

`ptc_solver__enum_get_position(Identifier, Value)`
Returns the possible values of an enumeration variable without knowing its base type
e.g. `ptc_solver__enum_get_position(Tommorow, Value)` Usage is not recommended in normal circumstances.

`ptc_solver__enum_get_basetype(Identifier, Basetype)`
Returns the Basetype of an enumeration variable Identifier
e.g. `ptc_solver__enum_get_basetype(Today, weekday) would succeed`

7.9.5 Error Messages

`ptc_solver__error(Message)`
Outputs the string Message as an error message and aborts the solver
e.g. `ptc_solver__error("problem line forty_two/2")` Usage is not recommended in normal circumstances. Write your own error reporting predicate