Cancer Research UK
Advanced Computation Laboratory

PRO*forma* technical paper

Title: Syntax and Semantics of PRO*forma*

Authors: David Sutton, John Fox
Version: 1.3.38
Status: Draft
Date: 26/03/2003

Summary

Describes the syntax and semantics for Pro*forma*

Contents

Part 1: Context Free Syntax

1. Introduction

A syntactically correct PRO*forma* guideline must conform to the BNF and lexical grammar laid out in part I of this document. Furthermore it must conform to the context sensitive constraints laid out in part II. Every expression must be type-correct in the sense that its inferred type (¶4.1) is not `untypeable`. Furthermore the use of identifiers within the guideline must satisfy the contextual constraints laid out in ¶6.

2. Lexical Grammar

PRO*forma*'s BNF syntax is defined in terms of the following lexical tokens

- A *reserved word* is any text string that appears in double quotes in the BNF productions set out in ¶3.2 . For example the appearance of the string "completed" in the BNF should be taken as indicating that the lexical analyser recognises the string `completed` a reserved word.

- An *atom* consists of either
  - A text string which consists of one or more underscores or non-whitespace alphanumeric characters and which does not begin with a digit, e.g. `pro1234_Forma`, or `_P`.
  - A pair of single quotes enclosing a sequence of zero or more characters which may contain any character other than an unescaped single quote, e.g. `'pRo \':: Ma'`

  Atoms are represented in the BNF by the symbol <*atom*>

- An *integer* is an optional minus sign ('-') followed by one or more digits. Integers are represented in the BNF by the symbol <*integer*>

- A *float* is an optional minus sign ('-') followed by **either**
  - a sequence of zero or more digits followed by a period ('.') followed by one or more digits, **or**
  - a sequence of one or more digits followed by a period ('.') followed by zero or more digits.

  optionally followed by a sequence consisting of one of the letters "e", "E", "d", or "D" and then one or more digits.

  Floats are represented in the BNF by the symbol <*float*>.

- A *double quoted string* is a pair of double quote characters enclosing a sequence of zero or more characters which may contain any character other than an unescaped double quote, e.g. `“pRo 'z”:: Ma”.` Double quoted strings are represented in the BNF by the symbol <*double_quoted_string*>

- *White Space* means any sequence of one or more spaces, newlines, carriage returns or tabs. White space is not represented in the BNF.

- A *comment* is the string `/**` followed by any sequence of characters that does not include the string `**/` followed by the string `**/.` For example

  ```
  /** This is a comment **/
  ```

  Comments are not represented in the BNF.

The lexical analyser converts ASCII text into tokens by starting at the beginning of the text and following this procedure:

1. Identify for the longest string that matches one of the lexical rules above, if the longest string matches more than rule then use the rule that comes first in the list above (e.g. the string `completed` is interpreted as a reserved word rather than an atom because the rule for reserved words comes first).
2. If the string identified was not white space and not a comment then add the appropriate token to the stream of tokens to be parsed by the BNF. For example if you have recognised an atom then add <*atom*> to the stream of tokens.
3. Unless the end of the text has been reached, start at the end of the string that has just been recognised and repeat steps 1,2,3.

3. BNF Syntax

3.1. Assumptions and Notation

The Backus-Naur Form (BNF) syntax of PRO*forma* 1.0 given in ¶3 assumes that the ASCII text defining a PRO*forma* guideline has been converted into lexical tokens by a lexical analyser whose grammar is defined in ¶2.

The following notational conventions are used in the BNF syntax:

a) Roman text strings enclosed in angle brackets, e.g. <task> denote non-terminal symbols.
b) Italic text strings enclosed in angle brackets, e.g. <*atom*> denote tokens recognised by the lexical analyser that are not reserved words.
c) Text strings enclosed in double quotes represent reserved words recognized by the lexical analyser.

The root symbol of the BNF syntax is <process>

3.2. BNF Productions

<abort_condition> ::= “abort” “::” <expression> “;”

<action_attribute> ::= <task_attribute>
<action_attribute> ::= <procedure>

<action_attribute_list> ::= <action_attribute> <action_attribute_list>

<action_attribute_list> ::= <empty>

<action_task >::= “action” “::” <*atom*> ";” <generic_attribute_list>
<action_attribute_list> “end” “action” “.”

<argument> ::= “argument” “::” <support> “,” <expression>
<optional_argument_attribute_list> “;”

<assertion> ::= <assignment>
<assertion> ::= <assertion> “and” <assertion>
<assertion> ::= “(“ assertion “)”

<assignment> ::= <*atom*> “=” <expression>

<autonomous> ::= “autonomous” “::” <yes_or_no> “;”

<candidate> ::= “candidate” “::” <candidate_name> “;”
<generic_attribute_list> <candidate_attribute_list>

<candidate_attribute> ::= <argument>
<candidate_attribute> ::= <recommendation>
<candidate_attribute> ::= <priority>

<candidate_attribute_list> ::= <candidate_attribute> <candidate_attribute_list>
<candidate_attribute_list >::= <empty>

<candidate_name> ::= <*atom*>

<caption> ::= “caption” “::” <expression> “;”

<choice_mode> ::= “choice_mode” “::” <choice_mode_type> “;”

<choice_mode_type> ::= “single”
<choice_mode_type> ::= “multiple”

<complex_atom> ::= <*atom*> “:” <*atom*>

<component> ::= “component” “::” <*atom*> “;” <component_attribute_list>

<component_attribute> ::= <autonomous>
<component_attribute> ::= <optional>
<component_attribute> ::= <terminal_att>
<component_attribute> ::= <param_value>
<component_attribute> ::= <schedule_constraint>
<component_attribute> ::= <ltwh>
<component_attribute> ::= <number_of_cycles>
<component_attribute> ::= <cycle_until>
<component_attribute> ::= <cycle_repeat>

<component_attribute_list> ::= <component_attribute> <component_attribute_list>
<component_attribute_list> ::= <empty>

<constant> ::= <number>
<constant> ::= <textual_constant>

<context> ::= “context” “::” <*atom*> “;”

<cycle_repeat> ::= “cycle_repeat” “::” <expression> <time_unit> “;”

<cycle_until> ::= “cycle_until” “::” <expression> “;”

<data_attribute> ::= <range>
<data_attribute> ::= <default_value>
<data_attribute> ::= <true_value>
<data_attribute> ::= <false_value>
<data_attribute> ::= <mandatory_validation>
<data_attribute> ::= <derivation>
<data_attribute> ::= <warning_condition>
<data_attribute> ::= <unit>

<data_attribute_list> ::= <data_attribute> <data_attribute_list>
<data_attribute_list> ::= <empty>

<data_item> ::= “data” “::” <data_name> “;” <data_type_definition>
<generic_attribute_list> <data_attribute_list> “end” “data” “.”

<data_name> ::= <*atom*>
<data_name> ::= <complex_atom>

<data_type> ::= <*atom*> (*must be either “text”, “integer”, “boolean”, “datetime”, “date”, “time”, “real”, “setof_text”, “setof_integer”, or “setof_real”)*

<data_type_definition> ::= “type” “::” <data_type> “;”

<decision_attribute> ::= <task_attribute>
<decision_attribute> ::= <candidate>
<decision_attribute> ::= <source>
<decision_attribute> ::= <choice_mode>
<decision_attribute> ::= <support_mode>

<decision_attribute_list> ::= <decision_attribute> <decision_attribute_list>
<decision_attribute_list> ::= <empty>

<decision_task> ::= “decision” “::” <*atom*> “;” <generic_attribute_list>
<decision_attribute_list> “end” “decision” “.”

<default_value> ::= “default_value” “::” <expression> “;”

<derivation> ::= “derivation” “::” <expression> “;”

<description> ::= “description” “::” <expression> “;”

<directive_list> ::= <*atom*> “;” <directive_list>
<directive_list> ::= <empty>

<empty> ::=

<enquiry_attribute> ::= <task_attribute>
<enquiry_attribute> ::= <source>

<enquiry_attribute_list> ::= <enquiry_attribute> <enquiry_attribute_list>
<enquiry_attribute_list> ::= <empty>

<enquiry_task> ::= “enquiry” “::” <*atom*> “;”
<generic_attribute_list> <enquiry_attribute_list>
“end” “enquiry” “.”

<expression> ::= <*atom*>
<expression> ::= <complex_atom>
<expression> ::= <*integer*>
<expression> ::= <*float*>
<expression> ::= <*double quoted string*>
<expression> ::= “(“ <expression> “)”
<expression> ::= “result_of” “(“ <*atom*> “)”
<expression> ::= “Netsupport” “(“ <*atom*> “,” <*atom*> “)”
<expression> ::= “netsupport” “(“ <*atom*> “,” <*atom*> “)”
<expression> ::= <expression> “or” <expression>
<expression> ::= <expression> “OR” <expression>
<expression> ::= <expression> “and” <expression>
<expression> ::= <expression> “AND” <expression>
<expression> ::= <expression> “#” <expression>
<expression> ::= <expression> “++” <expression>
<expression> ::= <expression> “<” <expression>
<expression> ::= <expression> “<=” <expression>
<expression> ::= <expression> “=<” <expression>
<expression> ::= <expression> “>” <expression>
<expression> ::= <expression> “>=” <expression>
<expression> ::= <expression> “=>” <expression>
<expression> ::= <expression> “=” <expression>
<expression> ::= <expression> “!=” <expression>
<expression> ::= <expression> “+” <expression>
<expression> ::= <expression> “-” <expression>
<expression> ::= <expression> “*” <expression>
<expression> ::= <expression> “/” <expression>
<expression> ::= <expression> “include” <expression>
<expression> ::= <expression> “includes” <expression>
<expression> ::= <expression> “oneof” <expression>

<expression> ::= “-” <expression>
<expression> ::= <functor_name> “(“ <expression_list> “)”
<expression> ::= “[“ expression_list “]”

<expression_list> ::= <empty>
<expression_list> ::= <nonempty_expression_list>

<false_value> ::= “false_value” “::” <textual_constant> “;”

<functor_name> ::= <*atom*>

<generic_attribute> ::= <caption>
<generic_attribute> ::= <description>

<generic_attribute_list> ::= <generic_attribute> <generic_attribute_list>
<generic_attribute_list> ::= <empty>

<generic_task> ::= “task” “::” <*atom*> “;” <generic_attribute_list>
<task_attribute_list> “end” “task” “.”

<goal> ::= “goal” “::” <expression> “;”

<ltwh> ::= “ltwh” “::” <*integer*> “,”<*integer*> “,”<*integer*> “,”<*integer*>“;”

<mandatory> ::= “mandatory” “::” <yes_or_no> “;”

<mandatory_validation> ::= “mandatory_validation” “::” <expression> “;”

<nonempty_expression_list> ::= <expression>
<nonempty_expression_list> ::= <expression> “,” <nonempty_expression_list>

<nonempty_parameter_list> ::= <parameter>
<nonempty_parameter_list> ::= <parameter> “,” <nonempty_parameter_list>

<number> ::= <*integer*>
<number> ::= <*float*>
<number> ::= “- ” <number>

<number_of_cycles> ::= “number_of_cycles” “::” <expression> “;”

<optional> ::= “optional” “::” <yes_or_no> “;”

<optional_argument_attribute_list> ::= <empty>
<optional_argument_attribute_list> ::= “attributes” <optional_argument_name>
<generic_attribute_list> “end” “attributes”

<optional_argument_name> ::= “argument_name” “::” <*atom*> “;”
<optional_argument_name> ::= <empty>

<optional_data_type_definition> ::= <data_type_definition>

<optional_data_type_definition> ::= <empty>

<optional_directives> ::= “directives” “::” <directive_list> “end” “directives” “.”
<optional_directives> ::= empty

<optional_parameter_attribute_list> ::= <empty>
<optional_parameter_attribute_list> ::= “attributes” <optional_data_type_definition>
<generic_attribute_list> “end” “attributes”

<param_value> ::= “param_value” “::” <assignment> “;”

<parameter> ::= <*atom*> <optional_parameter_attribute_list>

<parameter_declarations> ::= “parameters” “::” <nonempty_parameter_list> “;”

<plan_attribute> ::= <task_attribute>
<plan_attribute> ::= <component>
<plan_attribute> ::= <abort_condition>
<plan_attribute> ::= <terminate_condition>

<plan_attribute_list> ::= <plan_attribute> <plan_attribute_list>
<plan_attribute_list> ::= <empty>

<plan_task> ::= “plan” “::” <*atom*> “;” <generic_attribute_list plan_attribute_list>
“end” “plan” “.”

<postcondition> ::= “postcondition” “::” <assertion> “;”

<precondition> ::= “precondition” “::” <expression> “;”

<priority> ::= “priority” “::” < *integer*> “;”

<procedure> ::= “procedure” “::” <expression> “;”

<process> ::= optional_directives plan_task top_level_component_list

<range> ::= “range” “::” <range_list> “;”

<range_list> ::= <expression>
<range_list> ::= <expression> “,” range_list

<recommendation> ::= “recommendation” “::” <expression> “;”

<schedule_constraint> ::= “schedule_constraint” “::” “completed” “(“<*atom*>“)” “;”

<source> ::= “source” “::” data_name “;” source_attribute_list

<source_attribute_list> ::= <source_attribute> <source_attribute_list>
<source_attribute_list> ::= <empty>

<source_attribute> ::= <generic_attribute>
<source_attribute> ::= <mandatory>

<support> ::= “for”
<support> ::= “against”
<support> ::= “confirming”
<support> ::= “excluding”
<support> ::= <number>

<support_mode> ::= “support_mode” “::” <support_mode_type> “;”

<support_mode_type> ::= “symbolic”
<support_mode_type> ::= “numeric”

<task> ::= <plan_task>
<task> ::= <decision_task>
<task> ::= <action_task>
<task> ::= <enquiry_task>
<task> ::= <generic_task>

<task_attribute> ::= <precondition>
<task_attribute> ::= <wait_condition>
<task_attribute> ::= <postcondition>
<task_attribute> ::= <goal>
<task_attribute> ::= <trigger>
<action_attribute> ::= <context>
<task_attribute> ::= <parameter_declarations>

<task_attribute_list> ::= <task_attribute> <task_attribute_list>
<task_attribute_list> ::= <empty>

<terminal_att> ::= “terminal” “::” <yes_or_no> “;”

<terminate_condition> ::= “terminate” “::” <expression> “;”

<textual_constant> ::= <*double quoted string*>
<textual_constant> ::= <*atom*>

<time_unit> ::= “seconds”
<time_unit> ::= “minutes”
<time_unit> ::= “hours”
<time_unit> ::= “days”
<time_unit> ::= “weeks”

<top_level_component> ::= <task>
<top_level_component> ::= <data_item>

<top_level_component_list>::=<top_level_component> <top_level_component_list>
<top_level_component_list> ::= <empty>

<trigger> ::= “trigger” “::” <*atom*> “;”

<true_value> ::= “true_value” “::” textual_constant “;”

<unit> ::= “unit” “::” textual_constant “;”

wait_condition ::= “wait_condition” “::” <expression> “;”

<warning_condition> ::= “warning_condition” “::” <constant> “,” <expression> “;”
generic_attribute_list

<yes_or_no> ::= <*atom*> (*must be either “yes” or “no”*)

3.3. Note on <yes_or_no> and <data_type>

The BNF specifies that <yes_or_no> must be an atom and there is a “side condition” to the effect that this atom must be one of the strings “yes” or “no”.

The reason why we do not simply put in productions <yes_or_no> ::= “yes” and <yes_or_no> ::= “no” is that this would imply that “yes” and “no” were reserved words of the language (see ¶2) and hence could not be used in places where a reserved word would unacceptable,. Specifically this would mean that “yes” and “no” could not be used as names of decision candidates, which would be rather irksome.

Similarly <data_type> must be one of the atoms “text”, “integer”, “boolean”, “datetime”, “date”, “time”, “real”, “setof_text”, “setof_integer”, or “setof_real”. However these are not PRO*forma* reserved words.

Part II: Context Sensitive Syntax

4. Type Inference Rules

The type of a PRO*forma* <expression> *E* is either `text`, `integer`, `real`, `setof_text`, `setof_real`, `setof_integer`, `setof_anything`, `truth_value`, or `untypeable` and can be inferred using the rules laid out below.

An expression is type-correct iff its inferred type is not `untypeable`.

In the rules below italicised words in angle brackets, e.g. <*atom*> refer to terminal symbols of the BNF set out in ¶3.2 and roman strings in angle brackets, e.g. <parameter_declaration> refer to non-terminal symbols in that grammar.

4.1. Type Inference Rules for an Expression E

1. If *E* is of the form < *atom*> then
    a. If *E* is within the scope of a <parameter_declaration> of the form *Name* [":" *DataType*] ";" where *Name* = *E* and where the use of square brackets indicates that the data type is optional then
        i. If *DataType* is absent then the type of *E* is `text`.
        ii. Else the type of *E* is related to the *DataType* in the manner set out in ¶4.3.
    b. Else if there is a data item whose name is *E* then the type of *E* is determined by the type of that data item, according to the mapping laid out in ¶4.3.
    c. Else the type of *E* is `text`.

2. Else if *E* is of the form <*integer*> then the type of *E* is `integer`.
3. Else if *E* is of the form <*float*> then the type of *E* is `real`.
4. Else if *E* is of the form <*double_quoted_string*> then the type of *E* is `text`.
5. Else if *E* is of the form "(" *E¢* ")" where *E¢* is an <expression> then the type of *E* is the same as the type of *E¢*.
6. Else if *E* is of the form "result_of" "(" <*atom*> ")" then the type of *E* is `text`.
7. Else if *E* is of the form "netsupport" "(" <*atom*> "," <*atom*> ")" then the type of *E* is `integer`.
8. Else if *E* is of the form "[" "]" then the type of *E* is `setof_anything`.
9. Else if *E* is of the form "[" E_1 "," … "," E_n "]" where $n>0$ then
    a. If for all *i* such that $(1 \le i \le n)$ the expression E_i has type `text` then *E* has type `setof_text`.

b. Else if for all *i* such that $(1 \leq i \leq n)$ the expression E_i has type `integer` then *E* has type `setof_integer`.
c. Else if for all *i* such that $(1 \leq i \leq n)$ the type of E_i is either `integer` or `float` then the type of *E* is `setof_float`.
d. Otherwise *E* is of type `untypeable`.

10. Else if E is of either the form E_1 *Op* E_2 or the form *Op* "(" E_1 "," … "," E_n ")" where E_1 … E_n are <expression>s and *Op* is either an <infix_op> or a <functor_name> and $T_1 \ldots T_n$ respectively are the types of the expressions $E_1 \ldots E_n$ then
    a. If there exists a unique type T_{n+1} such that one of the allowed types for *Op* (see ¶11) is $(T_1 \times \ldots \times T_n) \rightarrow T_{n+1}$ then the type of E is T_{n+1}.
    b. Else if there exist types $T\!\!\downarrow_1 , \ldots , T\!\!\downarrow_n , T_{n+1}$ such that one of the allowed types for *Op* (see ¶11) is $(T\!\!\downarrow_1 \times \ldots \times T\!\!\downarrow_n) \rightarrow T_{n+1}$ and, for all i such that $(1 \le i \le n)$ either $T\!\!\downarrow_i = T_i$ or else T_i can be promoted (¶4.2) to $T\!\!\downarrow_i$ then the type of E is T_{n+1}. If *Op* has more than one type that meets this requirement, then use the one that comes first in its list of types.
    c. Else the type of E is `untypeable`.

4.2. Type Promotion Rules

1. `integer` can be promoted to `real`
2. `setof_integer` can be promoted to `setof_real.`
3. `setof_anything` can be promoted to `setof_text`, `setof_integer` or `setof_real`.
4. No other type promotions are possible.

4.3. Relationship Between Declared And Inferred Types For Data Items And Parameters.

The <data_type> that occurs in the declaration of a parameter or a data item has the syntax

<data_type> = <*atom*>

with the constraint that the <*atom*> must be one of "text", "integer", "boolean", "datetime", "date", "time", "real", "setof_text", "setof_integer", or "setof_real"

The relationship between the words "text", "integer" etc. as used in PRO*forma* type declarations and the types that are actually inferred for the parameters and data items is slightly complicated in that

a) The engine treats data items and parameters declared to be "boolean" in the same way as data items declared to be of type "text". A "boolean" data item is just a text data item for which the user is asked two choose between two different strings (e.g. "yes" and "no") when supplying a value.
b) Data items and parameters declared to be of type "date", "datetime" or "time" are treated by the engine in exactly the same way as if they were declared to be of type "real". The declared type is used as a hint to the API that the data should be entered by the user in some date/time format and then converted to a real number in an implementation dependent manner before being sent to the engine.

In summary the relationship between the declared type for data items/parameters and the types inferred for references to them are as follows:

Declared Type	Inferred Type
Text	`text`
Integer	`integer`
Boolean	`text`
Datetime	`real`
Date	`real`
Time	`real`
Real	`real`
setof_text	`setof_text`
setof_integer	`setof_integer`
setof_real	`setof_real`

5. Scope Rules

The following rules define the scopes of identifiers used to name tasks, data items, parameters, and candidates.

- The scope of a task name is global.
- The scope of a data item name is global.
- The scope of a candidate name is global.
- If a parameter name *P* is introduced in a <parameter_declaration> which is part of the <task> declaration for some task named *T* then *P* may be referred to in
    - The precondition of *T*.
    - The right hand side of an assignment in the postcondition of *T*.
    - The definition of an argument or recommendation rule for a candidate of *T* (if *T* is a decision).
    - The left hand side of a <param_value> assignment occurring within a <component> that instantiates *T*, i.e. a <component> that is of the form "component" "::" *T* ";" {<component_attribute> } See the example below.
    - The right hand side of a <param_value> occurring within the definition of a <component> of *T* (if *T* is a plan). See the example below.

If a parameter name *P* appears outside the scope of that parameter then it is interpreted as denoting the string *"P"* rather than the parameter *P* (see example below).

5.1. Example Of Use Of Parameters

The example below illustrates correct and incorrect use of parameter names.

```
plan :: Protocol35 ;
  caption :: 'Protocol35' ;
  component :: plan1 ;
    param_value :: _P =  1 ; /**legal since _P is a parameter of plan1 **/
end plan .

action :: action1 ;
  parameters :: _Q, _S, _T ;
  precondition :: _Q=1 ; /**legal since _Q is a parameter of action1**/
end action .

enquiry :: enquiry1 ;
  parameters :: _R ;
  source :: a ;
    mandatory :: yes ;
end enquiry .

plan :: plan1 ;
  parameters :: _P ;
  component :: action1 ;
     param_value :: _Q =  a ; /**correct _Q is a parameter of action1 and is thus
                                 allowed  to appear on the LHS of this
                                 assignment**/

     param_value :: _S = _T ; /**legal BUT  _T will be taken to denote the string
                                 "_T" since the scope of the parameter _T does
                                  include the RHS of this assignment **/

     param_value :: _P = 1 ; /**illegal!!  _P can appear on the RHS
                               but not the LHS**/

  component :: enquiry1 ;
    param_value :: _R =  _P ; /**legal _R is a parameter of enquiry1 and _Q is a
                                 parameter of plan1**/
end plan .

data :: a ;
  type :: integer ;
end data .
```

6. Contextual Constraints

In addition to conforming to the BNF and type system a valid PRO*forma* guideline must obey a set of contextual constraints. These concern the use of task names, candidate names and built-in operators and functions.

Definition 1: a string with the syntax <task> is referred to as a *task definition* for a task named *T* where *T* is the <*atom*> following the first occurrence of the reserved word "::" in that string. For example the string

action :: action2 ;
end action.

is a task definition for a task named action2.

Definition 2: a string with the syntax <candidate> is referred to as a *candidate definition* for a candidate named *C* where *C* is the <*atom*> following the first occurrence of the reserved word "::" in that string. For example the string

candidate :: mycand ;
    recommendation :: Netsupport(decision1, mycand) >= 1 ;

is a candidate definition for a candidate named mycand.

Definition 3: a string with the syntax <component> is referred to as a *component definition* for a component named *C* where *C* is the <*atom*> following the first occurrence of the reserved word "::" in that string. For example the string

component :: action2 ;

is a component definition for a component named action2:

6.1. Constraints On The Use Of Task, Candidate And Parameter Names.

1. If a plan definition contains a component definition for a component named *T* then the guideline must contain exactly one task definition for a task named *T*.

2. If an expression of the form "result_of" "(" *T* ")" occurs in anywhere in a guideline then the guideline must contain a task definition for a task named *T*.

3. If an expression of the form "netsupport" "(" *T* "," *C* ")" occurs anywhere in a guideline then the guideline must contain a task definition for a task named *T*

and that task definition must contain a candidate definition for a candidate named *C*.

4. If a component definition for a component named *T* contains a <param_value> assignment of the form *P* "=" *E* then the task definition for task *T* must include a <parameter> declaration for the parameter *P*.

6.2. Note On Use Of Built-In Functions and Operators.

The type constraints laid out in ¶4.1 ensure that wherever an <infix_op> or <functor_name> appears in an <expression> the operator or functor referred to will be one of the PRO*forma* built-ins listed in ¶11 and will be applied to the correct number of arguments. Hence there is no need to apply additional contextual constraints on the use of these operators.

Part III: Semantics

7. Basic Concepts

7.1. Guidelines, Tasks, and Data Items.

a) A guideline has a set of *guideline components.* Where no ambiguity can arise we shall refer to these simply as *components*

b) We assume that there exists an infinite set of *Component Identifiers.* These are abstract entities that can be used to uniquely identify components within a guideline

c) Each guideline component is one of a *Task*, a *Data Item* or a *Subcomponent.*

d) A Task is one of a *Generic task*, a *Plan*, a *Decision*, an *Enquiry*, or an *Action*.

e) A Subcomponent is one of a *Source*, a *Candidate*, a *Parameter*, or a *Warning Flag*.

f) The *class* of a component is one of `data_item`, `generic_task`, `plan`, `decision`, `enquiry`, `action`, `source`, `candidate`, `parameter`, or `warning_flag`. In other words the class tells you whether the component is a Data Item, a Task, or a Subcomponent and, in the last two cases, what sort of Task or Subcomponent it is.

g) A Guideline component has a set of named *properties*. The name of a property is a text string and the value that it has for a particular component is a PRO*forma* value (see section 7.4). As an example each task has a property named *description* whose value is a text string.

For each class of component there is a fixed set of allowable property names and for each property name and component class there are restrictions on the values that the property may take.

The properties of each class of component are set out in detail in ¶10.

Some of the properties of a component, such as its description, remain constant throughout the enactment of a guideline. Others, such as its state, change their value as enactment progresses.

Some properties are assigned initial values in the textual description of a PRO*forma* guideline. However other properties, such as the state of a task, are given values by the engine.

*7.2. PRO*forma *Expressions.*

a) A *PROforma expression* is a text string obeying the syntax given for <expression> in the PROforma BNF. Where no ambiguity can arise we shall refer to PRO*forma* expressions simply as *expressions*.

b) At any given time an expression has a *value*. The value of the expression may change as a guideline is executed. The value of an expression is a PRO*forma* value (as set out in ¶7.4).

The evaluation of expressions is explained in detail in ¶9.

c) Every expression has a *type*, which may be one of `text`, `integer`, `real`, `setof_text`, `setof_real`, `setof_anything`, `truth_value`, or `untypeable` The rules by which the type of an expression may be inferred are described in ¶4.

7.3. *PRO*forma Assertions

A PRO*forma* assertion is a text string having the syntax of <assertion>

*7.4. PRO*forma *Values*

A PRO*forma* value is one of:

1. The constant `unknown`.
2. One of the constants `true`, or `false`.
3. One of the constants `dormant`, `discarded`, `in_progress`, `completed`, `symbolic`, `numeric`, `multiple`, `single`, `for`, `against`, `confirming`, `excluding`, `data_item`, `generic_task`, `plan`, `decision`, `enquiry`, `action`, `source`, `candidate`, `parameter`, or `warning_flag`.
4. A number (either integer or floating point).
5. A text string (which may, in some cases be a PRO*forma* expression).
6. A Component Identifier
7. A finite sequence of PRO*forma* values. We use angle brackets to denote sequences, e.g. ⟨1,2,3,4,5⟩.

7.5. Task States

A task has a property named `state`, which may take the values `dormant`, `in_progress`, `discarded`, or `completed`.

The intended meaning of these states is as follows. A task is `dormant` if the PRO*forma* engine has not yet considered executing it, a task is `discarded` if the

engine has explicitly decided not to execute it, a task is `in_progress` if it is currently being executed, and a task is `completed` if its execution has finished.

Tasks change state when the operations such as Start (¶8.5.3), Discard (¶8.5.4) or Complete (¶8.5.5) are performed.

8. The Abstract PRO*forma* Engine.

In order to define the operational semantics of PRO*forma* we give here an abstract definition of the *state* of a PRO*forma* guideline and an abstract definition of how that state changes when various operations are performed.

The operational semantics are defined in terms of *operations* which change the guideline state, *conditions* which may, at any given time, be true or false, and *functions* which may be used to calculate a PRO*forma* value (¶7.4).

Operations, conditions, and functions may take parameters, which are usually either component identifiers or text strings. Operations, conditions, and functions may also refer to the current guideline state. It might be more formally correct to insist that the guideline state be passed as a parameter, however, in order to avoid cluttering our notation we shall not do so in this document.

Any actual implementation of a PRO*forma* engine must define itself in terms of the abstract engine. In other words the data returned by any method in the engine's API must have a precisely defined relationship to the abstract engine's state. And any state changes brought about by an API function must be definable in terms of the operations specified in the definition of the abstract engine.

The definition of the operational semantics describes six operations as being *public*. These are LoadGuideline, RunEngine, SetEngineTime, ConfirmTask, CommitCandidates, AddData The intention is that an API could be defined in terms of just these six operations. All the other operations in the description of the engines semantics exist solely in order to help define the public operations.

8.1. Guideline State

The *state* of a guideline is defined by following four data structures.

1. A *Properties* table, which contains the current values of the properties of the guideline components.
2. A *Changes* table, which contains a set of changes that have been requested to properties of guideline components.
3. A logical flag *Exception*, which is true if an abnormal event has occurred in the processing of a guideline operation and false otherwise.
4. A real number known as the *EngineTime.*
5. An real number known as the *RandomNum* which is used to generate random numbers.

Both the Properties and Changes tables have three columns; the first contains Component Identifiers, the second property names, and the third PRO*forma* values. We assume that the ordering of the rows in these tables is immaterial and that neither table may contain duplicate rows i.e. two different rows in a given table must have different values in at least one column.[1] The operational semantics guarantees that, at any given time the Properties table only assigns one value for a given property of a given component.

As an illustration the, tables in Example 1 indicate that the `state` of the task `action1` currently has the value `in_progress`, and that a request has been made to change the value of that task's `state` to `performed`.

Properties Table

Component ID	**Property**	**Value**
`action1`	`state`	`in_progress`
`action1`	`procedure`	`myprocedure`
.	.	.
.	.	.
.	.	.

Changes Table

Component ID	**Property**	**Value**
`action1`	`state`	`performed`

Example 1

.

In the sections that follow we shall explain how rows get added to the tables and what the intended meanings of those rows are.

8.2. Definitions

Definition 4: The *Task Set* of a guideline is the set Component Identifiers identifying tasks. More formally it is the set of all component identifiers C such that

[1] Mathematically speaking each table is a ternary relation. The Changes table is a subset of $C \times P \times V$ and the Properties table is a function which is a subset of $(C \times P) \rightarrow V$ where C is the set of component identifiers, P is the set of property names, and V is the set of PRO*forma* values.

(*C*,`class`,*T*) appears in the Properties Table where *T* is one of `generic_task`, `plan`, `enquiry`, `decision`, or `action`.

Definition 5: A *Task ID* is a Component Identifier that is a member of the guideline's Task Set.

Definition 6: The terms *Data Item Id, Candidate Id, Source Id, Parameter Id,* and *Warning Flag Id* refer to component identifiers and are defined in an analogous manner to the term Task Id.

8.2.1. Relationship to The PRO*forma* Syntax

The initial state of the guideline is related to its representation in PRO*forma* syntax by the description of the LoadGuideline Operation, which is given in ¶12.1.

8.3. Public Operations

An *Operation* is a process which can change the state of the guideline and which may take parameters.

This section lays out the *public* operations that the API to an implementation of PRO*forma* must make available to external systems. Sections ¶8.4 and ¶8.5 describes private operations used in the definition of the public operations.

In the following sections each operation, condition, and function referred to in the operational semantics is given a *definition* which is intended to precisely specify what it does by reference to the guideline state, and may be given a more informal *description.*

8.3.1. The LoadGuideline Operation

The LoadGuideline operation takes one parameter which is a text string obeying the syntax of <guideline> in the PROforma BNF. It is fully defined ¶12.1.

8.3.2. The RunEngine Operation

Parameters: none

Description:

The *Engine Cycle* is the basic means by which enactment of a PRO*forma* guideline proceeds. Its definition refers to two operations *Review* and *EnactChanges* which are respectively defined in ¶8.5.1 and ¶8.4.

Definition:

1. Set *RandomNum* to real value in the range 0 to 1 inclusive. We do not define how this value is chosen but the intention is that it should be generated in a pseudo-random manner.
2. For every task identifier *T* in the guideline's task set, perform the suboperation *Review*(*T*).
3. If the Changes table is empty then stop, otherwise perform the *EnactChanges* suboperation.
4. If the Exception flag is true then stop, otherwise repeat again from step 1.

Note: in step 1 we do not specify the order in which tasks are *Review*ed . However the definition of the engine's operational semantics in fact ensures that the state of the guideline after an engine cycle has been performed does not depend on the order in which tasks are *Review*ed during that engine cycle.[2]

8.3.3. The Operation SetEngineTime(*X*)

Parameters:

- *X* is a real number.

Description:

Update the *EngineTime.* Note that performing this operation is the *only* way to change the Engine Time. An application that wishes events to take place in "real" time must therefore make sure that the SetEngineTime method is performed before every invocation of RunEngine and that the time it sets is as close to the "real" time as possible.

Definition:

1. Set the *EngineTime* to *X*.

8.3.4. The Operation ConfirmTask(*T*)

Parameters:

- *T* is a task identifier.

Desciption:

The API performs the ConfirmTask operation when it has received confirmation from an external agent that a given task has actually been performed.

[2] I shall not prove this assertion in this document, but it is, I hope not to difficult to see that it is true.

Definition:

1. If the Properties Table contains a row (*T*, `confirmed`, *V*) for some *V* then remove that row.

2. Add (*T*, `confirmed`, `true`) to the Properties table.

8.3.5. The Operation CommitCandidates(*T*, <C_1, …, C_N>)

Parameters:

- *T* is a task Identifier, which should identify a decision
- <C_1, …, C_n> is a sequence of Component identifiers, each of which should identify one of the decision's candidates.

Description:

An API performs the CommitCandidates operation when an external agent indicates that it wishes to commit to one or more candidates of a decision.

Definition:

1. If the Properties Table contains a row (*T*, `result`, *R*) for some *R* then remove that row.

2. If the properties table contains (*T*, `choice_mode`, `multiple`) then add (*T*,`result`,⟨C_1, …, C_n⟩) to the Properties table, else if *n=1* then add (*T*,`result`, C_1) to the Properties table.

3. Perform ConfirmTask(*T*)

8.3.6. The Operation AddDataValue(*D*,*V*)

Parameters:

- *D* is a component identifier identifying either a data item or a parameter.
- *V* is a PRO*forma* value.

Definition:

1. If there is a value *V¢* such that the Properties table contains the row (*D*,`value`,*V¢*) then remove that row.

2. Add the rows (*D*,`value`,*V*) and (*D*,`requested`,`false`) to the Properties table.

3. If there is an expression *E* such that the Properties table contains (*D*,`mandatory_validation`, *E*) then remove any row of the form (*D*,`mandatory_validation_activated`, *V*) from the the Properties table and replace it with (*D*,`mandatory_validation_activated`, *V*¢) where *V*¢=*Evalutate*(*E*,*R*) and *R* is the component identifier of the root plan of the guideline.

4. If there is a sequence of component identifiers ⟨*W*1, … , *Wn*⟩ such that the Properties table contains (*D*,`warning_conditions`, ⟨*W*1, … , *Wn*⟩) then for each *I* such that i £*i*£*n* :

    a. Find the unique expression *E* such that the Properties table contains (*Wi*, `expression`, *E*)

    b. Remove any row of the form (*Wi*, `activated`, *V*) from the the Properties table and replace it with (*Wi*,`activated`, *V*¢) where *V*¢=*Evalutate*(*E*,*R*) and *R* identifies the root plan of the guideline.

    c. Perform ActualiseGenericProperties(*W*,*R*) where *R* is the component identifier of the root plan of the guideline.

8.3.7. The Operation SendTrigger(*Trig*)

Parameters:

- *Trig* is a text string

Desciption:

Set `trigger_active` property to true for all tasks with the specified trigger whose parent plans are in progress.

Definition:

For each task identifier *T* such that the Properties Table contains (*T*, `trigger`, *Trig*) and there exists a task identifier *P* such that the Properties table contains (*T*,`parent_plan`,*P*) and (*P*,`state`,`in_progress`) : add (*T*,`trigger_active`, `true`) to the Properties table.

8.4. The Operation EnactChanges

Parameters: none

Description:

The *EnactChanges* operation performs the changes that get requested during the performance of the RunEngine operation described in ¶8.3.2 .

Definition:

1. For every row in the Changes table:

    a. Let (C,P,V) denote the values in the Component ID, Property, and Value columns of that row.
    b. If there exists a value $V¢$ such that the Properties table contains the row $(C,P,V¢)$ then remove that row.

    c. If (C,P,V) is the only row in the Changes table that specifies a new value for property P of component C then add (C,P,V) to the Properties table.

    d. Otherwise, i.e. if there exists a value $V¢¢$ such that $V¢¢ \neq V$ and the Changes table contains $(C,P,V¢¢)$, then set the *Exception* flag to true and add $(C,P,\text{unknown})$ to the current properties table.

2. Empty the Changes table.

8.5. Operations Taking a Task Identifier As A Parameter

8.5.1. The Operation Review(*T*)

Parameters:

- *T* is a task identifier.

Description:

The RunEngine operation (¶8.5.1) performs the *Review* operation for each task in the guideline in order to determine what that task should do.

The definition of Review(*T*) refers to operations Initialise, Start, Discard, and Complete which are defined in sections 8.5.2,8.5.4, 8.5.5, and 8.5.3, and to the conditions InitialiseConditions, StartConditions , DiscardConditions , and

CompleteConditions which are defined in sections8.6.1, 8.6.2, 8.6.3 and **Error! Reference source not found.**8.6.4
Definition:

1. If InitialiseConditions(*T*) is true then perform Initialise(*T*)
2. Else if StartConditions(*T*) is true then perform Start(*T*)
3. Else if DiscardConditions(*T*) is true then perform Discard(*T*)
4. Else if CompleteConditions(*T*) is true then perform Complete(*T*)
5. Else do nothing.

8.5.2. The Operation Initialise(*T*)

Parameters:

- *T* is a task identifier.

Definition:

1. Add (*T*, `state`, `dormant`) to the Changes table.
2. Add (*T*, `cycle_count`, `0`) to the Changes table.
3. Add (*T*, `start_at`, `unknown`) to the Changes table.
4. Add (*T*, `trigger_active`, `false`) to the Changes table.
5. Add (*T*,`nbr_cycles_value`,`unknown`) to the Changes table.
6. Perform InitialiseGenericProperties(*T*,*T*).
7. If the Properties table contains a row (*T*, `parameters`, <P_1 ,…, P_N>) then for each P_i (1≤*i*≤*N*):
   - Add (P_i,value,`unknown`) to the Changes table.
   - Perform InitialiseGenericProperties(P_i,*T*).
8. If the Properties table contains (*T*,`class`,`decision`) then:
   a. Add (*T*,`result`,`unknown`) to the Changes table.
   b. If the Properties table contains a row (*T*, `candidates`, <C_1 ,…, C_N>) then for each C_i (1≤*i*≤*N*) perform the operation InitialiseCandidate(C_i).
9. If the Properties table contains (*T*,`class`,`action`) then:
   a. Add (*T*,`actual_procedure`,`unknown`) to the Changes table.

8.5.3. The Operation Start(*T*)

Parameters:

- *T* is a task identifier.

Description:

The Start operation sets a task's state to `in_progress`, increments its `cycle_count`, evaluates its parameters and initialises various other properties of the task such as its sources.

It is important that to note that parameter values are evaluated within the scope of the task's parent. This is because parameter values are a property of the parent plan rather than the task itself and it allows a parameter value assignment to refer to the parameters of the parent plan.

Definition:

1. Add (*T*, `state`, `in_progress`) to the Changes table.

2. Add (*T*, `start_at`, `unknown`) to the Changes table.

3. (*T*,`in_progress_time`, *EngineTime*) to the Changes table.

4. If the Properties table contains a row (*T*, `cycle_count`, `N`) then add (*T*,`cycle_count`,`N+1`) to the Changes table.

5. If the Properties table contains a row (*T*, `parameters`, $<P_1 ,\ldots, P_N>$) then for each P_i ($1 \leq i \leq N$):

    - If there exists an expression *E* and a component identifier C_P such that the Properties table contains the rows (P_i,expression,*E*) and (*T*,parent_plan,C_P) then let *V*=Evaluate(*E*,C_P) and add (P_i,value,*V*) to the Changes table.
    - Otherwise add (P_i,value,`unknown`) to the Changes table.
    - Perform ActualiseGenericProperties(P_i,*T*).

6. If there exists an expression *E* such that the Properties table contains the row (*T*,nbr_cycles_expression,*E*) and there does not exist a real number *R* such that the properties table contains (*T*,nbr_cycles_value,*V*) then:

    - Let *V*= Evaluate(*E*,*T*) and add (*T*,nbr_cycles_value,*V*) to the Changes table.

7. Perform the operation ActualiseGenericProperties(*T*,*T*).

8. Add (*T*, `confirmed`, `false`) to the Changes table.

9. If the Properties table contains (*T*, `class`, `enquiry`) and (*T*,`sources`,<S_1,…,S_N>) then, for each S_i perform the operation InitialiseSource(S_i) as define in ¶8.5.12.

10. If the Properties table contains (*T*, `class`, `decision`) and (*T*,`sources`,<S_1,…,S_N>) then, for each S_i :

    - If the properties table contains a row (S_i, `data_item`, *D*) where *D* is a component identifier and also contains a row (*D*,`value`,`unknown`) then perform the operation InitialiseSource(S_i) as defined in ¶8.5.12.

    - If the Properties table contains a row (*T*,`candidates`,<C_1,…,C_N>) then for each C_i ($1 \le i \le N$) perform the operation InitialiseCandidate(C_i).

11. If the Properties table contains (*T*, `class`, `decision`) then

    - If the Properties table contains (*T*,`confirmatory`,`false`):
        - Find the identifiers of recommended candidates of *T* (if any). The definition of a "recommended" candidate is given below.
        - If there are one or more recommended candidates then let *C* be the identifier of the recommended candidate with the highest netsupport (if there is more than one candidate with this netsupport value then pick the one with the highest priority). Netsupport is calculated using the function EvaluateNetSupport defined in ¶9.4.
        - Add (*T*,`result`,*C*) to the Changes table.
    - Otherwise add (*T*,`result`,`unknown`) to the Changes table

12. If the Properties table contains (*T*, `class`, `action`) then

    - If the Properties table contains a row (*T*, procedure,*E*)where *E* is a PRO*forma* expression then let *V*=Evaluate(*E*,*T*) as defined in ¶9.1, and add the row

      (*T*, actual_procedure, *V*)

      to the Changes table.

    - Else add (*T*, actual_procedure, `unknown`) to the Changes table.

Definition of a Recommended Candidate:

A candidate identifier *C* identifies a *recommended* candidate of task *T* iff there exists an expression *E* and a sequence of candidate identifiers $\langle C_1, \ldots, C_n \rangle$ such that:

- The Properties table contains (*T*,`candidates`,$\langle C_1, \ldots, C_n \rangle$) and (*C*,`recommendation`,*E*) **and**
- Evaluate(*E*,*T*)=`true` (see ¶9.1), **and**
- There exists an *i* such that ($1 \le i \le n$) and $C_i = C$.

8.5.4. The Operation Discard(*T*)

Parameters:

- *T* is a task identifier.

Definition:

1. Add (*T*,`state`,`discarded`) to the Changes table.
2. Perform the operation ActualiseGenericProperties(*T*,*T*).
3. (*T*,`discarded_time`, *EngineTime*) to the Changes table.

8.5.5. The Operation Complete(*T*)

Parameters:

- *T* is a task identifier.

Definition:

1. Add (*T*,`state`,`completed`) to the Changes table.
2. (*T*,`completed_time`, *EngineTime*) to the Changes table.
3. If there exists an <assertion> *A* such that the Properties table contains (*T*,postcondition,*A*) then perform EnactAssertion(*A*,*T*), as defined in ¶8.5.11.
4. If CycleConditions(*T*) is true then perform SetStartAt(*T*)
5. If CycleConditions(*T*) is not true then add (*T*,`start_at`,`unknown`) to the Changes table.

6. If the Properties Table contains a row (T,`sources`,<S_1,…,S_N>) then, for each S_i such that the Properties table contains a rows (S_i, `data_item`, D) and (D,`requested`,`true`) and (D,`default_value`, V) :

    - If V≠`unknown` then add the rows (D,`value`, V) and (D,`requested`,`false`) to the Changes table.

7. If the Properties table contains (T,`class`,`decision`) and (T,`confirmatory`,`false`) then

    a. Let C_1,…,C_n be the recommended candidates of T (see ¶8.5.3).

    b. If the Properties table contains (T,`choice_mode`,`multiple`) then add the Row (T, `result`, ⟨C_1,…,C_n⟩) to the Changes Table.

    c. Otherwise add (T, `result`, ⟨C_i⟩) to the Changes Table where C_1 is the identifier of the recommended candidate with the highest netsupport (if there is more than one candidate with this netsupport value then pick the one with the highest priority). Netsupport is calculated using the function EvaluateNetSupport defined in ¶9.4.

    Note: the definition of Complete Conditions (¶8.6.4) ensures that a decision that completes must have at least one recommended candidate.

8. If the Properties table contains (T,`class`,`decision`) and also contains a row (T, `candidates`, <C_1 ,…, C_N>) then for each C_i (1≤i≤N) perform the operation ActualiseCandidate(C_i,T)

8.5.6. The Operation SetStartAt(*T*)

Parameters:

- T is a component identifier identifying a task.

Description:

Sets the time at which a cycling task should restart.

Definition:

- If there exists an expression E and a text string U such that the Properties table contains (T,`cycle_interval`, E) and (T,`cycle_interval_units`, U) then

    a. Let M = 1000 if U= “seconds”, 1000*60 if U= “minutes”, 1000*60*60 if U= “hours”, 1000*60*60*24 if U= “days”, and 1000*60*60*24*7 if U= “weeks”, and 0 otherwise.

b. Let V = Evaluate(E,T)

c. add (T,`start_at`,*EngineTime*+V*M) to the Changes table.

- Else add (T,`start_at`, *EngineTime*) to the Changes table.

8.5.7. The Operation InitialiseCandidate(*C*)

Parameters:

- C is a component identifier identifying a candidate of a decision.

Description:

Sets the `actual_caption` and `actual_description` of the candidate and to `unknown`. Does the same for the `actual_captions` and `actual_descriptions` of the candidate's arguments.

Definition:

1. Perform InitialiseGenericProperties(C).
2. If the Properties Table contains a row (C, `arguments`, <A_1 ,…, A_N>) then for each A_i ($1 \le i \le N$) perform the operation InitialiseGenericProperties(A_i).

8.5.8. The Operation ActualiseCandidate(*C*,*T*)

Parameters:

- C is a component identifier identifying a candidate of a decision.
- T is a task identifier identifying the decision that owns C.

Description:

Sets values for the `actual_caption` and `actual_description` of the candidate by evaluating its `caption` and `description`. Does the same for the `actual_captions` and `actual_descriptions` of the candidate's arguments.

Definition:

1. Perform ActualiseGenericProperties(C).
2. If the Properties Table contains a row (C, `arguments`, <A_1 ,…, A_N>) then for each A_i ($1 \le i \le N$) perform the operation ActualiseGenericProperties(A_i).

8.5.9. The Operation InitialiseGenericProperties(*C*)

Parameters:

- *C* is a component identifier.

Description:

Sets the `actual_caption` and `actual_description` of the component to `unknown`.

Definition:

1. Add (*C*,`actual_caption`,`unknown`) to the Changes table
2. Add (*C*,`actual_description`,`unknown`) to the Changes table

8.5.10. The Operation ActualiseGenericProperties(*C*,*T*)

Parameters:

- *C* is a component identifier.
- *T* is a task identifier identifying the task that owns *C*.

Description:

Sets values for the `actual_caption` and `actual_description` of the component by evaluating its `caption` and `description`.

Definition:

1. If there exists an expression E_c such that the Properties table contains the row (*T*,`caption`, E_c) then let V_c=Evaluate(E_c,*T*) and add (*T*,`actual_caption`,V_c) to the Changes table

2. If there exists an expression E_d such that the Properties table contains the row (*T*,`description`, E_d) then let V_d=Evaluate(E_d,*T*) and add (*T*,`actual_description`, V_d) to the Changes table

8.5.11. The Operation EnactAssertion(*A*,*T*)

Parameters:

- *A* is a text string whose syntax is defined by <assertion>

- *T* is a task identifier identifying the task to which this assertion belongs

Definition:

1. If *A* is of the form *Name* = *E* where *Name* is an <*atom*> and *E* is an <expression> then
    - Let *D* = ResolveDataReference(*Name*) as defined in ¶9.5.
    - Let *V* = Evaluate(*E*,*T*), as defined in ¶9.1.
    - Add (*D*, value, *V*) to the changes table.

2. Else if *A* is of the form A_1 "and" A_2 where A_1 and A_2 are <assertion>s then perform EnactAssertion(A_1,*T*) and then perform EnactAssertion(A_2,*T*).

8.5.12. The Operation InitialiseSource(*S*,*T*)

Parameters:

- *S* is a component identifier identifying a source.

Definition:

- If there exists a data item identifier *D* such that the Properties table contains the row (*S*,`data_item`,*D*) but does not contain the row (*D*,`requested`,`true`) then let *R* be the component identifier of the root plan of the guideline:
    - Add the row (*D*,`requested`,`true`) to the Changes table.
    - Perform the Operation ActualiseGenericProperties(*S*,*R*).
    - Perform the Operation ActualiseGenericProperties(*D*,*R*).
    - If there exists a sequence of expressions $\langle E_1, \ldots, E_n \rangle$ such that the properties table contains (*D*,`range_expressions`,$\langle E_1, \ldots, E_n \rangle$) then add (*D*,`range_values`,$\langle V_1, \ldots, V_n \rangle$) to the changes table, where V_1 = Evaluate(E_1,*R*) for ($1 \leq i \leq n$) and where *R* identifies the root plan of the guideline.
    - If there exists an expression *E* such that the properties table contains (*D*,`default_expression`, *E*) then add (*D*,`default_value`, *V*) to the changes table, where *V* = Evaluate(*E*,*R*).

8.6. Task Conditions

The following conditions can be true or false for a given task identifier *T*.

8.6.1. InitialiseConditions(*T*)

Parameters:

- *T* is a task identifier.

Description:

The InitialiseConditions of a task are true if it is a triggered task which has just completed or if the CycleConditions or InitialiseConditions of the task's parent plan are true.

Definition:

InitialiseConditions(*T*) is true iff either:

1. The Properties table contains (*T*,`state`,`completed`) and (*T*,`trigger_active`,`true`) and does not contain (*T*,`terminal`,`true`), **or**

2. There exists a task identifier *P* such that the Properties table contains (*T*,`parent_plan` *P*) **and** Either StartConditions(*P*) or InitialiseConditions(*P*) is true.

8.6.2. StartConditions(*T*)

Parameters:

- *T* is a task identifier.

Description:
The StartConditions of a task are true iff, its parent plan (if any) is in_progress and either; its ScheduledStartConditions, are true and it has no trigger; or it has been triggered or has completed and needs to begin a new cycle.

Definition:

StartConditions(*T*) is true iff:

1. If there exists a component identifier *P* such that the Properties table contains (*T*,`parent_plan`, *P*) then the Properties table also contains (*P*,`state`,`in_progress`) , **and**

2. One of the following conditions hold:

    a. ScheduledStartConditions(*T*) is true and there does **not** exist a string S such that the Properties table does contains (*T*,`trigger`,*S*), **or**

b. The Properties table contains (*T*,`trigger_active`,`true`) and (*T*,`state`,`dormant`), **or**
c. There exists a real number *R* such that the properties table contains (*T*,`start_at`, *R*) and *R* is less than or equal to the current engine time.

8.6.3. DiscardConditions(*T*)

Parameters:

- *T* is a task identifier.

Description:

The DiscardConditions of a task are true iff either

1. It is either dormant or in_progress and the DiscardConditions of its parent plan are true, **or**

2. It is currently dormant, its parent plan is in_progress, its ScheduleConditions are true, and either it has antecedent tasks that have all been discarded, or it has a precondition that is not true.

Definition:

Discard Conditions(*T*) is true iff either of conditions 1, 2 or 3 below are true

1. There exists a component identifier *P* such that the Properties table contains (*T*,`parent_plan`, *P*) and (*P*,`state`,`in_progress`), **and**

    a. The Properties Table contains either (*T*,`state`,`in_progress`), or (*T*,`state`,`dormant`), or both (*T*,`state`,`completed`) and (*T*,`start_at`,*R*) for some real number *R*, **and either**

        i. DiscardConditions(*P*) is true, **or**

        ii. TerminationConditions(P) is true.

Or;

2. Conditions a,b, and c below are true

    a. Either:
        i. There does not exists a component identifier *P* such that the Properties table contains (*T*,`parent_plan`, *P*) **or**

        ii. There exists a component identifier *P* such that the Properties table contains (*T*,`parent_plan`, *P*) and (*P*,`state`,`in_progress`),

    And

    b. The Properties table contains (*T*,`state`,`dormant`)

And

c. ScheduleConditions(*T*) is true **and either:**
   i. There exists a non-empty sequence of task identifiers $\langle T_1,\ldots,T_n\rangle$ such that the Properties table contains (*T*,`antecedent_tasks`,$\langle T_1,\ldots,T_n\rangle$) and for all i such that $1\leq i\leq n$ the Properties table contains (T_i,`state`,`discarded`); **or**
   ii. There exists an expression *E* such that the Properties table contains (*T*,`precondition`,*E*) and Evaluate(*E*,*T*)≠`true`.

Or;

3. The Properties Table contains (*T*,`state`,`in_progress`) **and** there is an expression *E* such that the Properties table contains (*T*,`abort_condition`,*E*) **and** Evaluate(*E*,*T*)=`true`

8.6.4. CompleteConditions(*T*)

Parameters:

- *T* is a task identifier.

Description:

The CompleteConditions of a task are true if it is currently in progress and has been confirmed (if it is confirmatory), and has been supplied with values for all of its mandatory sources (if it is a decision or enquiry) and if all of its mandatory components or at least one of its terminal components has completed (if it is a plan) and if none of its components is in_progress or could subsequently become in_progress or be discarded (if it is a plan).

Definition:

CompleteConditions(*T*) is true iff the following conditions are all true:

1. The Properties table contains (*T*,`state`,`in_progress`); **and**

2. If there exists a sequence of Source identifiers ⟨S_1,…,S_n⟩ such that the Properties table contains (*T*, `sources`, ⟨S_1,…,S_n⟩) then for all *i* such that (1≤*i*≤*n*):
    - If there exists a Data Item identifier *D* such that the Properties table contains (*S*$_i$,`data_item`,*D*) and (*S*$_i$,`mandatory`,`true`) then it also contains (*D*,`requested`,`false`);

    and

3. If the Properties table contains (*T*,`confirmatory`,`true`) then it also contains (*T*,`confirmed`,`true`) ; **and**

4. If the Properties table contains (*T*,`class`,`decision`) and (*T*,`confirmatory`,`false`) then *T* has at least one recommended candidate ((see ¶8.5.3); **and**

5. If the Properties table contains (*T*,`class`,`plan`) then
    b. For all task identifiers *C* such that the Properties table contains (*C*,`parent_plan`,*T*) :

        i. The Properties table contains either (*C*,`optional`,`true`) (*C*,`state`,`completed`) or (*C*,`state`,`discarded`) , **and**
        ii. The Properties table does not contain (*C*,`state`,`in_progress`) , **and**
        iii. StartConditions(*C*) is false, **and**
        iv. DiscardConditions(*C*) is false, **and**
        v. InitialiseConditions(C) is false, **and**
        vi. There does not exist a real number *R* such that the Properties table contains (*C*,`start_at`,*R*)

8.6.5. ScheduledStartConditions(*T*)

Parameters:

- *T* is a task identifier.

Description:
The Scheduled StartConditions of a task are true iff it is currently dormant, it ScheduleConditions are true, at least one of its antecedent tasks (if it has any) has completed, and its precondition (if any) is true.

Definition:

ScheduledStartConditions(*T*) is true iff the following three conditions are true:

1. The Properties table contains (*T*,`state`,`dormant`); **and**

2. ScheduleConditions(*T*) is true **and**

3. If there exists a non empty sequence of task identifiers $\langle T_1,\ldots,T_n\rangle$ such that the Properties table contains (*T*,`antecedent_tasks`, $\langle T_1,\ldots,T_n\rangle$) then there exists an i such that $1\leq i\leq n$ and the Properties table contains (T_i,`state`,`completed`); **and**

4. If there exists an expression *E* such that the Properties table contains (*T*,`precondition`,*E*) then Evaluate(*E*,*T*)=`true`.

8.6.6. ScheduleConditions(*T*)

Parameters:

- *T* is a task identifier.

Description:

The ScheduleConditions of a task are true iff all of its antecedent tasks have either completed or been discarded.

Definition:

The ScheduleConditions(*T*) is true iff the following conditions are true:

1. If there exists a sequence of task identifiers $\langle T_1,\ldots,T_n\rangle$ such that the Properties table contains (*T*,`antecedent_tasks`, $\langle T_1,\ldots,T_n\rangle$) then for all i such that $1\leq i\leq n$:
    a. the Properties table contains either (T_i,`state`,`completed`) or (T_i,`state`,`discarded`), **and**
    b. There does not exist a real number *R* such that the Properties table contains (T_i,,`start_at`,*R*).

    And

2. If there exists an expression *E* such that the properties table contains (*T*,`wait_condition`,*E*) then Evaluate(*E*,*T*)=`true`.

8.6.7. CycleConditions(*T*)

Parameters:

- *T* is a task identifier.

Description:

The CycleConditions of a task are true iff either has a `cycle_until` condition which has not yet been satisfied or a `nbr_cycles_value` property whose value is greater than that of its `current_cycle` property.

Definition:

CycleConditions(*T*) is true iff

1. There exist integers *I,C* such that the Properties table contains (*T*,`current_cycle` ,*I*) and (*T*,`nbr_cycles_value`,*N*) and *N* > *C* ; **and**
2. If there exists an expression *E* such that the Properties table contains (*T*,`cycle_until`,*E*) then Evaluate(*E* , *T*)≠`true`.

8.6.8. TerminationConditions(*T*)

Parameters:

- *T* is a task identifier that identifies a Plan.

Description:

The TerminationConditions of a plan are true if either it's `termination_condition` exists and evaluates true or if it has a terminal task which has completed.

Definition:

TerminationConditions(*T*) is true if **either**

1. There is an expression *E* such that the Properties Table contains (*T*,`termination_condition`,*E*) and Evaluate(*E*,*T*)=`true`, **or**

2. There exists a task identifier *T2* such that the Properties Table contains (*T2*,`parent_plan`,*T*) and (*T2*,`state`,`completed`) .

9. Evaluation of Expressions

In the PRO*forma* language all expressions are "attached" to a task. If they appear in the preconditions, postconditions, or parameter value definitions or candidate definitions of a task then they are "attached" to that task. Expressions appearing in the

definition of data items are "attached" to the root plan. Hence the function Evaluate, which returns the value of an expression, takes two parameters specifying the expression and the task to which it is attached.

9.1. The Function Evaluate(E,T)

Parameters:

- *E* is a text string with the syntax <expression>
- *T* is a component identifier identifying the task within which the expression *E* occurs.

Value: Evaluate(*E*,*T*) evaluates to a PRO*forma* value.

Definition:

- If *E* is an <*atom*> then:
  - If there is a component identifier *D* such that *D*=ResolveDataReference(*E*,*T*) then Evaluate(*E*,*T*)=EvaluateDataReference(*D*,*T*) as defined in ¶9.2.
  - Otherwise Evaluate(*E*,*T*) = the text string *E* with any enclosing single quotes removed.
- If *E* is a <number> then Evaluate(*E*,*T*) = the numeric value of *E*, i.e. the value you get by converting the text string *E* to an integer or floating point number using the usual conventions.
- If *E* is a <*double_quoted_string*> then Evaluate(*E*,*T*) = the text string *E* with the enclosing double quotes removed.
- If *E* is of the form "(" E_1 ")" where E_1 is an <expression> then Evaluate(*E*,*T*) = Evaluate(E_1,*T*).
- If *E* is of the form "result_of" "(" *A* ")" where *A* is an <*atom*> then
  - If T_A=ResolveTaskReference(*A*,*T*), as defined in ¶9.6, and there exists a candidate identifier *C* such that the Properties table contains the rows (T_A,result,*C*) and (*C*,name,*V*) then Evaluate(*E*,*T*)=*V*.
  - Else Evaluate(*E*,*T*)=unknown.
- Else if *E* is of the form "netsupport" "(" A_1 "," A_2 ")" where A_1 and A_2 are <atom>s then Evaluate(*E*,*T*)=EvaluateNetSupport(T_1,*C*), as defined in ¶9.4 where T_1=ResolveTaskReference(A_1,*T*), as defined in ¶9.6, and *C*=ResolveCandidateReference(A_2, T_1), as defined in ¶9.8.
- If *E* is of the form E_1 *Op* E_2 where E_1 and E_2 are <expression>s and *Op* is an <infix_op> then Evaluate(*E*,*T*)=F_{op}(V_1 , V_2) where V_i = Evaluate(E_i ,*T*) for ($1 \le i \le n$) and F_{op} is the evaluation function for *Op*, as defined in ¶11.

- If E is of the form Op "(" E_1 "," … "," E_n ")" where Op is a <functor_name> and E_i is an <expression> (for each i such that $1 \leq i \leq n$) then Evaluate(E,T)=F_{op}(T,V_1 ,…, V_N) where V_i = Evaluate(E_i ,T) and F_{op} is the evaluation function for Op, as defined in ¶11.

- If E is a <set_enumeration> of the form "[" E_1 "," … "," E_n "]" where E_i is an <expression> (for each i such that $1 \leq i \leq n$) then Evaluate(E,T) = the sequence <Evaluate(E_1,T), … , Evaluate(E_n,T)>.

9.2. The Function EvaluateDataReference(D,T)

Parameters:

- D is a component identifier that identifies a parameter or data item referred to in some expression.
- T is a component identifier identifying the task to which the expression that refers to D belongs.

Value: EvaluateDataReference(D,T) evaluates to a PRO*forma* value.

Definition:

1. If the Properties table contains the row (D, class, `parameter`) then EvaluateDataReference(D,T) = EvaluateParameter(D,T) as defined in ¶9.3.

2. Else if there exists a value V such that the Properties table contains the rows (D,class,`data_item`) and (D,value,V) then EvaluateDataReference(D,T)=V.

3. Else EvaluateDataReference(D,T)=`unknown`.

9.3. The Function EvaluateParameter(D,T)

Parameters:

- D is a component identifier that identifies a parameter that is referred to in some expression.
- T is a component identifier identifying the task to which the expression that refers to D belongs.

Value: EvaluateParameter(D,T) evaluates to a PRO*forma* value.

Description:

EvaluateParameter is used to evaluate the value of a task's parameter. There are a couple of subtleties to bear in mind when considering parameter evaluation. The first

is that the values of a task's parameters get fixed when the task becomes `in_progress` but that it is useful to be able to refer to them in the task's preconditions, which are evaluated while the task is still `dormant`. The semantics given here mean that if a parameter is encountered during the evaluation of a precondition then it will be given the value that it would have if the task became `in_progress` at that moment. The second subtlety is that the expression that defines the parameters value is part of the specification of the task's parent plan, and consequently is evaluated within the scope of that parent plan.

Definition:

1. If there exists a component identifier *P* and an expression *E* such that the Properties table contains the rows (*T*,parent_plan,*P*), (*D*,expression,*E*) and (*T*,state,`dormant`) then EvaluateDataReference(*D*,*T*) = Evaluate(*E*,*P*).
2. Else if there exists a value *V* such that the Properties table contains the rows (*D*,value,*V*) and (*T*,state,`in_progress`) then EvaluateDataReference(*D*,*T*)=*V*.
3. Else EvaluateDataReference(*D*,*T*)=`unknown`.

9.4. The Function EvaluateNetSupport(T,C)

Parameters:

- *T* is a task identifier
- *C* is a component identifier identifying a candidate

Value: EvaluateNetSupport(*T*,*C*) evaluates to an integer or to the constant `unknown`.

Definition:

- If there exists a sequence ⟨*A1*, … , *An*⟩ such that the Properties table contains a row (*C*,`arguments`,⟨*A1*, … , *An*⟩) then for all *i* such that(1≤*i*≤*n*) let E_i be the unique expression such that the Properties table contains the row (A_i,`expression`,E_i), let S_i be the unique value such that the Properties table contains the row (A_i,`support`,S_i) and let V_i=Evaluate(E_i,*C*), then:
    - If there exist *i*,*j* such that V_i= V_j=`true` and S_i=`confirming` and S_j=`excluding` then EvaluateNetSupport(*T*,*C*)=`unknown`.
    - Else if there exists *i* such that V_i= `true` and S_i=`confirming` then EvaluateNetSupport(*T*,*C*)=9999.
    - Else if there exists *i* such that V_i= `true` and S_i=`excluding` then EvaluateNetSupport(*T*,*C*)=-9999.
    - Else EvaluateNetSupport(*T*,*C*)=W_1 + … + W_n where for all *i* such that(1≤*i*≤*n*):
        - If V_i= `true` and S_i=`for` then W_i = 1.
        - If V_i= `true` and S_i=`against` then W_i = -1.
        - If V_i= `true` and S_i is an integer then W_i = S_i.
        - If V_i≠ `true` then W_i = 0.
- Else EvaluateNetSupport(*T*,*C*)=`unknown`.

9.5. The Function ResolveDataReference(A,T)

Parameters:

- *A* is an <*atom*>
- *T* is a task identifier

Value: ResolveDataReference(*A*,*T*) evaluates to either the constant `unknown` or to a component identifier, which identifies a parameter or data item.

Definition:

1. If the properties table contains rows (*T*, `parameters`, <*P1*, … , P_N >) and there is an *A¢* such *A¢* and *A* are equal if case is ignored and a P_i such that $1 \le i \le n$ and and such that the Properties table contains (P_i,`name`, *A¢*) then ResolveDataReference(*A*,*T*) = P_i.

2. Else if there is an *A¢* such *A¢* and *A* are equal if case is ignored and a component identifier *D* such that the Properties table contains the rows (*D*, class, `data_item`) and (*D*, name, *A¢*) then ResolveDataReference(*A*,*T*) = *D*.

3. Else ResolveDataReference(*A*,*T*) = `unknown`.

*9.6. The Function ResolveTaskReference(*A,T*)*

Parameters:

- *A* is an <*atom*> or text string.
- *T* is a task identifier

Value: ResolveTaskReference(*A*,*T*) evaluates to either the constant `unknown`, or to a component identifier, which identifies a task.

Description:

The function ResolveTaskReference is used to determine whether a given atom *A* appearing in the description of a task *T* should be treated as a reference to some other task, and if so which task. The rule used is that if there is only one task named *A* then *A* is taken to refer to that task, other wise if there is only one task named *A* that belongs to the same plan as *T* then *A* refers to that task, otherwise *A* does not refer to a task at all.

Definition:

1. If there exists an *A¢* such *A¢* and *A* are equal if case is ignored and a *unique* task identifier C_A such that the Properties table contains the row (C_A, name,*A¢*) then ResolveTaskReference(*A*,*T*) = C_A.

2. Else if there exist an *A¢* such *A¢* and *A* are equal if case is ignored and *unique* task identifiers *P* and C_A such that:

    a. The Properties table contains the rows (*P*, parent_plan, *T*) and (C_A,name, *A¢*), **and**

b. IsAncestor(P, C_A) is true (see ¶9.7)

Then ResolveTaskReference(A,T) = C_A.

3. Else ResolveTaskReference(A,T) = unknown.

9.7. The Condition IsAncestor(T_1, T_2)

Parameters:

- T_1 and T_2 are task identifiers.

Description:

This condition is true iff T_1 identifies a plan and T_2 identifies a component task of T_1 or a component task of a component task of T_1 and so on recursively.

Definition:

1. If the Properties table contains the row (T_2 ,parent_plan, T_1) then IsAncestor(T_1, T_2) is true.

2. Else if there exists a task identifier T_3 such that IsAncestor(T_1, T_3) is true and the Properties table contains the row (T_2 ,parent_plan, T_3) then IsAncestor(T_1,T_2) is true.

3. Else IsAncestor(T_1,T_2) is false.

9.8. ResolveCandidateReference(A,T)

Parameters:

- A is an <*atom*>
- T is a task identifier

Value: ResolveDataReference(A,T) evaluates to either the constant unknown or to a component identifier, which identifies a candidate.

Definition:

- If the properties table contains a rows (T, candidates, <C_1, … , C_n>) and there is an $A¢$ such $A¢$ and A are equal if case is ignored and a C_i such that the Properties table contains (C_i,Name, $A¢$) then ResolveCandidateReference(A,T)= C_i.

- Else ResolveCandidateReference(*A*,*T*) = `unknown`.

10. Properties of Components

We list here the properties that each class of component may have, along with the allowed values for that property and its intended meaning. The semantics of PRO*forma* are such that a property never gets assigned a value that is not allowed. Consequently there is no need for the EnactChanges operation (¶8.4) to check that property values specified in the Changes table are actually allowed. The constant `unknown` is an allowed value of all properties.

10.1. Properties Generic To All Components

Property Name	**Allowed Values (besides `unknown`)**	**Intended Meaning**
`Class`	`data_item`, `plan`, `decision`, `enquiry`, `action`, `source`, `candidate`, `parameter`, or `warning_flag`	What sort of component it is.
`Name`	Any text string	A name for the component
`caption`	Any PRO*forma* expression of type `text`.	Expression that can be used to generate the component's caption.
`actual_caption`	Any text string.	Actual caption of the component, generated by evaluating the `caption` at the appropriate time.
`description`	Any PRO*forma* expression of type `text`.	An expression that can be used to generate a longer description of the component
`actual_description`	Any text string.	Actual description of the component, generated by evaluating the `description` at the appropriate time.

10.2. Properties Generic To All Tasks

Tasks may have all the generic component properties listed in ¶10.1 and in addition may have the following properties.

Property Name	Allowed Values (besides `unknown`)	Intended Meaning
`state`	One of `dormant`, `discarded`, `in_progress`, or `completed`.	The state of the task.
`antecedent_tasks`	A sequence of task identifiers.	The tasks that must be completed or discarded before this one starts.
`goal`	Any Truth Valued PRO*forma* expression.	What the task is intended to do.
`precondition`	Any Truth Valued PRO*forma* expression.	A condition that may be assumed to be true when the task starts.
`wait_condition`	Any Truth Valued PRO*forma* expression.	A condition that must be true before the task can start or be discarded.
`trigger`	A text string conforming to the syntax of <trigger> as laid out in the PRO*forma* BNF.	The task's trigger.
`trigger_active`	`true` or `false`	Whether or not the task's trigger is active.
`postcondition`	A PRO*forma* assertion	A condition that may be assumed to be true when the task completes.
`parameters`	A sequence of Component Identifiers each of which identifies one of the task's parameters.	The task's parameters.
`parent_plan`	A Task Identifier	The task's parent plan. For the root plan this property has the value `unknown`.
`confirmatory`	`true` or `false`	Whether or not the task needs to be confirmed.
`confirmed`	`true` or `false`	Whether or not the task has been confirmed.

`current_cycle`	Any positive integer	The current cycle number.
`nbr_cycles_value`	Any positive integer	The number of times this task should be performed once started.
`nbr_cycles_expression`	Any integer valued PRO*forma* expression.	Expression used to calculate the `nbr_cycles_value` property. Should evaluate to a number ≥ 1.
`cycle_until`	Any truth valued PRO*forma* expression.	Task should be repetitively performed until this condition is true.
`cycle_interval`	Any real valued PRO*forma* expression	Amount of time in to wait between cycles of this task. See also `cycle_interval_units`
`cycle_interval_units`	Either `seconds`, `minutes`, `hours`, `days`, or `weeks`.	Units in which the cycle interval is expressed.
`start_at`	Any floating point number.	Task will wait until Engine Time exceeds or equals this value and then start the task (used in cycling).
`optional`	`true` or `false`	If `true` this task must complete or be discarded before its parent plan completes (unless a terminal task completes).
`terminal`	`true` or `false`	If `true` then if this task completes then its parent plan also completes.
`lwth`	A sequence of four integers.	Cartesian coordinates of a point at which to display an icon representing this task in a GUI.
`context`	Any text string	Any additional information that may need to be provided when an action's actual procedure is performed, e.g. who is supposed to perform it and where.
`in_progress_time`	Any floating point number.	The engine time at which the task last entered the `in_progress` state. `unknown` if the task has not yet entered that state.
`discarded_time`	Any floating point number.	The engine time at which the task last entered the `discarded` state. `unknown` if the task has not

		yet entered that state.
completed_time	Any floating point number.	The engine time at which the task last entered the completed state. unknown if the task has not yet entered that state.

10.3. Properties Of Plans

A plan may have all the generic properties of tasks (¶10.2) in addition to the following properties.

Property Name	Allowed Values (besides unknown)	Intended Meaning
terminate_condition	Any truth valued PRO*forma* expression.	A condition that, if true, will cause the plan to terminate.
abort_condition	Any truth valued PRO*forma* expression.	A condition that, if true, will cause the plan to be aborted.

10.4. Properties Of Decisions

A decision may have all the generic properties of tasks (¶10.2) in addition to the following properties.

Property Name	Allowed Values (besides unknown)	Intended Meaning
candidates	A sequence of Candidate Identifiers	The decision's candidates.
sources	A sequence of Source Identifiers	The decision's sources.
support_mode	symbolic or numeric	Whether arguments are to be weighed up numerically or symbolically. Note that the definition of net support given in ¶9.4 attaches numeric values to symbolic weights such as for and against thus the distinction between numeric and symbolic weighting does not effect the semantics given in this document.
choice_mode	multiple or single	Whether many candidates may be chosen or only one.
result	A sequence of	The chosen candidate(s).

	candidate identifiers. If the `choice_mode` is `single` then this sequence contains only one identifier.	

10.5. Properties Of Actions

An action may have all the generic properties of tasks (¶10.2) in addition to the following properties.

Property Name	Allowed Values (besides `unknown`)	Intended Meaning
`procedure`	Any PRO*forma* expression of type `text`.	Statically defined procedure to be requested by the task.
`actual_procedure`	Any text string	Procedure that the task is actually requesting.

10.6. Properties Of Enquiries

An enquiry may have all the generic properties of tasks (¶10.2) in addition to the following properties.

Property Name	Allowed Values (besides `unknown`)	Intended Meaning
`sources`	A sequence of source identifiers.	The enquiry's sources.

10.7. Properties Of Data Items

Data items may have all the generic component properties listed in ¶10.1 and in addition may have the following properties.

Property Name	Allowed Values (besides `unknown`)	Intended Meaning
`type`	One of `text`, `integer`, `boolean`, `datetime`, `date`, `time`, `real`, `setof_text`, `setof_integer`, `setof_real`	The type of the data item
`value`	Any PRO*forma* value	The value that has been assigned to this data item.
`range_values`	Any sequence of text strings or numbers.	Allowed values for the data item.
`range_expressions`	Any sequence of PRO*forma* expressions.	Expressions that are used to calculate the `range_values` property.
`default_value`	Any text string or number	A default value to be suggested when requesting a value for this data item.
`mandatory_validation`	Any truth valued PRO*forma* expression.	A condition that must be true at the moment that a new value is supplied for this data item.
`warning_conditions`	A sequence of warning condition identifiers.	Warning conditions for this data item.
`derivation_rule`	Any PRO*forma* expression.	An expression used to calculate the value of this data item at the moment it becomes requested.
`unit`	Any text string	The units in which this data item's value is expressed.
`requested`	Either `true` or `false`.	Whether or not a new value has been requested for the data item.

10.8. Properties Of Candidates

Candidates may have all the generic component properties listed in ¶10.1 and in addition may have the following properties.

Property Name	Allowed Values (besides `unknown`)	Intended Meaning
`recommendation`	Any PRO*forma* expression	Condition that must be true in order for this candidate to be "recommended"
`priority`	Any integer	Priority of this candidate
`arguments`	Any sequence of argument identifiers.	Arguments associated with this candidate.

10.9. Properties Of Arguments

Arguments may have all the generic component properties listed in ¶10.1 and in addition may have the following properties.

Property Name	Allowed Values (besides `unknown`)	Intended Meaning
`support`	Either `confirming`, `excluding`, `for` `against`, or an integer or real number.	The support that this argument, if true, will add to its candidate.
`expression`	Any PRO*forma* expression	An expression defining this argument.

10.10. Properties Of Parameters

Parameters may have all the generic component properties listed in ¶10.1 and in addition may have the following properties.

Property Name	Allowed Values (besides `unknown`)	Intended Meaning
`value`	Any PRO*forma* value	The value that has been assigned to this parameter.
`expression`	Any PRO*forma* expression	An expression that will be evaluated in order to assign a value to this parameter.

10.11. Properties Of Sources

Sources may have all the generic component properties listed in ¶10.1 and in addition may have the following properties.

Property Name	Allowed Values (besides `unknown`)	Intended Meaning
`data_item`	A data item identifier	Data Item for which a value is to be provided.
`mandatory`	`true` or `false`.	Whether or not a value *must* be supplied.

10.12. Properties Of Warning Conditions

Warning conditions may have all the generic component properties listed in ¶10.1 and in addition may have the following properties.

Property Name	Allowed Values (besides `unknown`)	Intended Meaning
`expression`	Any PRO*forma* expression	The condition we want to be warned about.
`activated`	`true` or `false`.	Whether or not the `expression` evaluated to true when a value was last added to the warning condition's data item.

11. PRO*forma* Built-in Operators

This section lists the functors and infix operators that may occur in a PRO*forma* <expression> along with their types and evaluation functions, which are used by the function Evaluate (¶9.1) to evaluate the applications of these functions and operators.

11.1. Infix Operators

11.1.1. Arithmetic Operators "+", "-","*"

We give here the allowed types and evaluation rule for "+". The types for "-","*" are the same as that for "+". The evaluation rules for "-","*", "/" are identical to that for "+" except that in the definition of F_+ the arithmetic operator + is replaced by – and so on. Note that "-" is also the name of the unary minus operator (¶11.2.2).

Types for "+":

- `(integer×integer)→integer`
- `(real×real)→real`

Note that this typing allows an arithmetic operator to be applied to arguments whose types are `real×integer` or `integer×real` because we can promote the integer argument to real.

Definition of the evaluation function $F_{+}(T,V_1, V_2)$

- If V_1 and V_2 are both integers or reals then $F_{+}(V_1, V_2)= V_1 + V_2$
- Else if either V_1 =`unknown` or V_2 =`unknown` then $F_{+}(V_1, V_2)$ =`unknown`

11.1.2. Arithmetic Operator "/"

The only difference between the definition of "/" and that for "+", "-", and "*" is that the result of a division is always treated as real even if its arguments are integers.

Types for "/":

- `(real×real)→real`

Note that type promotion means that "/" is still typeable if one or both of its arguments are integers.

Definition of the evaluation function $F_{/}(T,V_1, V_2)$

- If V_1 and V_2 are both integers or reals then $F_{/}(V_1, V_2)= V_1 / V_2$
- Else if either V_1 =`unknown` or V_2 =`unknown` then $F_{+}(V_1, V_2)$ =`unknown`

11.1.3. Comparison Operators ">", "<",">=""=>", "<=", "=<", "=", "!=", "<>"

We give here the allowed types and evaluation rule for ">", "<",">=""=>", "<=", "=<", "=", "!=", "<>"

Note that all these operators return `false` if one or both comparands has the value `unknown`. This means that one cannot necessarily assume that not(a=b) is equivalent to (a != b).

Types for ">", "<",">=""=>", "<=", "=<", "=", "!=", "<>".

- `(real×real)→truth_value`
- `(text×text)→truth_value`
- `(setof_real× setof_real)→truth_value`
- `(setof_text×setof_text)→truth_value`

Definition of the evaluation function for ">""<", "<=", "=>", ">=", "=>" , "=", "<>"and "!="

The evaluation function for these operators is defined in terms of a function *comp*, *head*, and *tail* which we define below. Note that *comp*, *head*, and *tail* are not PRO*forma* built-in operators, we have simply introduced it in order to define the comparison operator.

For a non-empty sequence of values $\underline{V}$:

- *head*($\underline{V}$) is the first element in the sequence
- *tail*($\underline{V}$) is the rest of the sequence (an empty sequence if the $\underline{V}$ contains only one element).

Examples:

head(⟨1 ,2,3⟩) = 1
tail(⟨1 ,2,3⟩) = ⟨2,3⟩
head(⟨ "a" ⟩) = "a"
tail(⟨ "a" ⟩) = ⟨⟩

The function *comp* returns 1,-1,0,or `unknown` and is defined as follows:

- If V_1 = `unknown` or V_1 = `unknown` then *comp* (V_1, V_2) = `unknown`.

- Else if V_1 and V_2 are both integers or reals then
  - If $V_1 > V_2$ *comp* (V_1, V_2)= 1
  - Else if $V_1 < V_2$ *comp* (V_1, V_2)= -1
  - Else if $V_1 = V_2$ *comp* (V_1, V_2)= 0

- Else if V_1 and V_2 are both text strings then
  - If V_1 is lexicographically greater than V_2 ignoring considerations of case then *comp* (V_1, V_2)= 1
  - Else if V_1 is lexicographically less than V_2 ignoring considerations of case then *comp* (V_1, V_2)= -1
  - Else if V_1 is lexicographically equal to V_2 ignoring considerations of case then *comp* (V_1, V_2)= 0

- Else if V_1 and V_2 are non-empty sequences of values then

  - If *comp* (*head*(V_1),*head*(V_2))= 1 then *comp* (V_1, V_2)= 1
  - Else if *comp* (*head*(V_1),*head*(V_2))= -1 then *comp* (V_1, V_2)= -1
  - Else *comp* (V_1, V_2)= *comp* (*tail*(V_1),*tail*(V_2))

- Else if V_1 and V_2 are both empty sequences then *comp* (V_1, V_2)= 0
- Else if V_1 is an empty sequence then *comp*(V_1, V_2) = -1
- Else if V_2 is an empty sequence then *comp*(V_1, V_2) = +1

The evaluation functions for all comparison operators are defined in terms of *comp* as follows:

comp (V_1, V_2)	1	-1	0	unknown
$F_{>}(V_1, V_2)$	true	false	false	false
$F_{<}(V_1, V_2)$	false	true	false	false
$F_{>=}(V_1, V_2)$	true	false	true	false
$F_{<=}(V_1, V_2)$	false	true	true	false
$F_{=}(V_1, V_2)$	false	false	true	false
$F_{!=}(V_1, V_2)$	true	true	false	false

We treat ">=" and "=>" as synonyms, similarly "<=" is synonymous with "=<" and "<>" with "!=".

11.1.4. Boolean Operators "and", "or"

We give here the allowed types and evaluation functions for "and" and "or".

Types for "and":

- (truth_value×truth_value)→truth_value

Definition of the evaluation function $F_{and}(T, V_1, V_2)$

- If V_1 =true and V_2=true then $F_{and}(V_1, V_2)$= true
- Else $F_{and}(V_1, V_2)$= false.

Types for "or":

- (truth_value×truth_value)→truth_value

Definition of the evaluation function $F_{or}(V_1, V_2)$

- If either V_1 =true or V_2=true then $F_{or}(V_1, V_2)$= true
- Else $F_{or}(V_1, V_2)$= false.

11.1.5. Text Concatenation Operator "#"

Types for "#":

- `(text×text)→text`
- `(real×text)→text`
- `(text×real)→text`
- `(real×real)→text`

Definition of the evaluation function $F_{\#}(T, V_1, V_2)$

- If $V_1 \neq$ `unknown` and $V_2 \neq$ `unknown` then $F_{\#}(V_1, V_2)$=the concatenation of the textual representations of V_1 and V_2. The text representation of a text item is the item itself, the text representation of an integer or real is implementation dependent. As example $F_{\#}$("he", "llo")= "hello" and $F_{\#}$("pi is ","3.14159") might be "pi is 3.14159" depending on how the implementation represents real numbers as text.

- Else $F_{\#}(V_1, V_2)$= `unknown`.

11.1.6. Membership Operators "includes", "include", "oneof"

We give here the allowed types and evaluation function for "includes". The operator "include" is a synonym for "includes". The operator "oneof" is identical to "includes" except that the order its operands is reversed, i.e. E_1"oneof"$E_2 \equiv E_2$"includes"E_1

Types for "includes":

- `(setof_text×text)→truth_value`
- `(setof_real×real)→truth_value`

Definition of the evaluation function $F_{includes}(T, V_1, V_2)$

The definition uses the function *comp* introduced in ¶11.1.3 and is as follows:

- If V_1 is a sequence of values $\langle V_{1,1}, \ldots, V_{1,n} \rangle$ and if there exists a j such that $(1 \leq j \leq n)$ and $comp(V_{1,j}, V_2) = 0$ then $F_{includes}(V_1, V_2)$=`true`.
- Else $F_{includes}(V_1, V_2)$=`false`.

11.2. Prefix Functors

11.2.1. Conditional Operator "if"

We give here the allowed types and evaluation functions for "if".

Types for "if":

- (truth_value×truth_value×truth_value)→truth_value
- (truth_value×integer×integer)→integer
- (truth_value×real×real)→real
- (truth_value×text×text)→text

- (truth_value×setof_integer×setof_integer)
  →setof_integer

- (truth_value× setof_real×setof_real)→ setof_real
- (truth_value× setof_text×setof_text)→setof_text

Definition of the evaluation function $F_{if}(T, V_1, V_2, V_3)$

- If V_1 =true then $F_{if}(V_1, V_2, V_3) = V_2$
- Else If V_1 =false then $F_{if}(V_1, V_2, V_3) = V_3$
- Else $F_{if}(V_1, V_2, V_3)$ = unknown

11.2.2. Unary Minus Operator "-"

We give here the allowed types and evaluation rule for the unary minus operator "-".

Types for "-" (unary minus):

- (integer)→integer
- (real)→real

Definition of the evaluation function $F_{-}(T, V_1)$ for unary minus.

- If V_1 is an integer or real then $F_{-}(V_1) = -V_1$
- Else if either V_1 =unknown then $F_{-}(V_1)$ =unknown

11.2.3. The functor "isknown"

Types for "isknown":

- text→truth_value
- real→truth_value
- setof_text→truth_value
- setof_real→truth_value
- truth_value→truth_value

Definition of the evaluation function $F_{isknown}(T, V)$

- If V =`unknown` then $F_{isknown}(V)$=`true`.
- Else $F_{isknown}(V)$=`false`.

11.2.4. Boolean Operator "not"

We give here the allowed types and evaluation functions for "not"

Types for "not":

- `truth_value`→`truth_value`

Definition of the evaluation function $F_{not}(T, V)$

- If V =`false` then $F_{not}(V)$= `true`
- Else $F_{not}(V)$= `false`.

11.2.5. Operator "count"

Types for "count":

- `setof_real`→`integer`
- `setof_text`→`integer`

Definition of the evaluation function $F_{count}(T, V)$

- If V = is a non-empty sequence of values $\langle V_1, \ldots, V_n \rangle$ then $F_{count}(V)= n$
- Else if V = is an empty sequence then $F_{count}(V)= 0$
- Else $F_{count}(V)$= `unknown`

11.2.6. Operator "sum"

Types for "sum":

- `setof_integer`→`integer`
- `setof_real`→`real`

Definition of the evaluation function $F_{sum}(T, V)$

- If V = is a non-empty sequence of values $\langle V_1, \ldots, V_n \rangle$ and each value V_i $(1 \le i \le n)$ is a real number or integer then $F_{sum}(V)= V_1+\ldots+V_n$

- Else if $V=$ is an empty sequence then $F_{sum}(V)= 0$

- Else $F_{sum}(V)=$ `unknown`

11.2.7. Operator "max"

Types for "max":

- `setof_integer→integer`
- `setof_real→real`
- `setof_text→text`

Definition of the evaluation function $F_{max}(T, V)$

$F_{max}(T, V)$ is defined using the *comp* function introduced in ¶11.1.3.

- If $V=$ is a non-empty sequence of values $\langle V_1 , \dots , V_n \rangle$ and there exists an integer i such that all of the following conditions are true:

  - $1 \le i \le n$
  - $V_i \neq$ `unknown`
  - For all j such that $1 \le j \le n$: $comp(V_i , V_j) \ge 0$ or $comp(V_i , V_j) =$ `unknown`

  then $F_{max}(T, V)= V_i$.

- Else $F_{max}(T, V)=$ `unknown`.

11.2.8. Operator "min"

Types for "min":

- `setof_integer→integer`
- `setof_real→real`
- `setof_text→text`

Definition of the evaluation function $F_{min}(T, V)$

$F_{min}(T, V)$ is defined using the *comp* function introduced in ¶11.1.3.

- If $V=$ is a non-empty sequence of values $\langle V_1 , \dots , V_n \rangle$ and there exists an integer i such that all of the following conditions are true:

  - $1 \le i \le n$
  - $V_i \neq$ `unknown`

- For all j such that $1 \leq j \leq n$: $comp(V_i, V_j) \leq 0$ or $comp(V_i, V_j) = \texttt{unknown}$

then $F_{min}(T, V) = V_i$.

- Else $F_{min}(T, V) = \texttt{unknown}$

11.2.9. Operator "nth"

Types for "sum":

- `integer→setof_integer→integer`
- `integer→setof_real→real`
- `integer→setof_text→text`

Definition of the evaluation function $F_{nth}(T, V_1, V_2)$

- If V_2 is a sequence of values $\langle V_{2,1}, \ldots, V_{2,n} \rangle$ and V_1 is an integer i such that $1 \leq i \leq n$ then $F_{nth}(T, V_1, V_2) = V_{2,i}$.

- Else $F_{nth}(T, V_1, V_2) = \texttt{unknown}$.

11.2.10. Operators "is_dormant", "is_in_progress", "is_discarded" and "is_completed".

We give here the type and evaluation rule for is_dormant. The types for is_in_progress, is_discarded, is_completed are the same and their evaluation rules are identical except that the word "dormant" should respectively replaced by "in_progress", "discarded" and "completed".

Types for "is_dormant"

- `text→truth_value`

Definition of the evaluation function $F_{is\_dormant}(T, V)$

- Let C=ResolveTaskReference(V,T).
- If $C = \texttt{unknown}$ then $F_{is\_dormant}(T, V) = \texttt{unknown}$.
- Otherwise $F_{is\_dormant}(T, V) = \texttt{true}$ if the properties table contains (C,`state`,`dormant`) and $F_{is\_dormant}(T, V) = \texttt{false}$ otherwise.

11.2.11. Operators “in_progress_time”, “discarded_time” and “completed_time”.

We give here the type and evaluation rule for in_progress_time. The types for discarded_time, and completed_time are the same and their evaluation rules are identical except that the word “dormant” should respectively replaced by “in_progress”, “discarded” and “completed”.

Types for “in_progress_time”

- `text→real`

Definition of the evaluation function $F_{in\_progress\_time}(T, V)$

- Let C=ResolveTaskReference(V,T).
- If there exists a real number N such that the properties table contains (C,`in_progress_time`,N) then $F_{in\_progress\_time}(T, V)=N$ otherwise $F_{in\_progress\_time}(T, V)$=`unknown`.

11.2.12. Operator “union”

Types for “union”:

- `setof_anything→setof_anything→setof_anything`
- `setof_integer→setof_integer→setof_integer`
- `setof_real→setof_real→setof_real`
- `setof_text→setof_text→setof_text`

Definition of the evaluation function $F_{union}(T, V_1,V_2)$

- If V_1 is a sequence of values $\langle V_{1,1}, \ldots, V_{1,n}\rangle$ and V_2 is a sequence of values $\langle V_{2,1}, \ldots, V_{2,n}\rangle$ then $F_{union}(T, V_1,V_2) = \langle V_{1,1}, \ldots, V_{1,n}, V_{2,1}, \ldots, V_{2,n}\rangle$.

- Else $F_{union}(T, V_1,V_2)$ = `unknown`.

11.2.13. Operator “diff”

Types for “diff”:

- `setof_anything→setof_anything→setof_anything`
- `setof_integer→setof_integer→setof_integer`
- `setof_real→setof_real→setof_real`
- `setof_text→setof_text→setof_text`

Definition of the evaluation function $F_{diff}(T, V_1,V_2)$

- If V_1 is a sequence of values $\langle V_{1,1}, \ldots, V_{1,n} \rangle$ and V_2 is a sequence of values $\langle V_{2,1}, \ldots, V_{2,n} \rangle$ then $F_{diff}(T, V_1, V_2)$ = the result of taking V_1 and removing all values $V_{1,i}$ where $V_{1,i} \neq$ `unknown` and there exists j such that $V_{1,i} = V_{2,j}$.

- Else $F_{diff}(T, V_1, V_2)$ = `unknown`.

11.2.14. Operator "intersect"

Types for "intersect":

- `setof_anything→setof_anything→setof_anything`
- `setof_integer→setof_integer→setof_integer`
- `setof_real→setof_real→setof_real`
- `setof_text→setof_text→setof_text`

Definition of the evaluation function $F_{intersect}(T, V_1, V_2)$

- If V_1 is a sequence of values $\langle V_{1,1}, \ldots, V_{1,n} \rangle$ and V_2 is a sequence of values $\langle V_{2,1}, \ldots, V_{2,n} \rangle$ then $F_{intersect}(T, V_1, V_2)$ = the result of taking V_1 and removing all values $V_{1,i}$ where either $V_{1,i}$ = `unknown` or there does not exist j such that $V_{1,i} = V_{2,j}$.

- Else $F_{intersect}(T, V_1, V_2)$ = `unknown`.

11.2.15. Operator "abs"

Types for "abs":

- `integer→integer`
- `real→real`

Definition of the evaluation function $F_{abs}(T, V)$

- If V is an integer or real number then $F_{abs}(T, V)$=the absolute value of V.

- Else $F_{abs}(T, V)$== `unknown`.

11.2.16. Operator "exp"

Types for "exp":

- `real→real`

Definition of the evaluation function $F_{exp}(T, V)$

- If V is an integer or real number then $F_{exp}(T, V) = e^V$.
- Else $F_{exp}(T, V)$== `unknown`.

11.2.17. Operator "ln"

Types for "ln":

- `real→real`

Definition of the evaluation function $F_{ln}(T, V)$

- If V is an integer or real number and $V>0$ then $F_{ln}(T, V)$= the natural logarithm of V.
- Else if V is an integer or real number and $V£0$ then $F_{ln}(T, V)$ is undefined and an attempt to evaluate it will set the *Exception* flag to `true`.
- Else $F_{ln}(T, V)$== `unknown`.

11.2.18. Operator "sin"

Types for "sin":

- `real→real`

Definition of the evaluation function $F_{sin}(T, V)$

- If V is an integer or real number and then $F_{sin}(T, V)$= the trigonometric sine of V (i.e. we assume V is expressed in radians).
- Else $F_{sin}(T, V)$= `unknown`.

11.2.19. Operator "cos"

Types for "cos":

- `real→real`

Definition of the evaluation function $F_{cos}(T, V)$

- If V is an integer or real number and then $F_{cos}(T, V)$= the trigonometric cosine of V (i.e. we assume V is expressed in radians).

- Else $F_{cos}(T, V)$= `unknown`.

11.2.20. Operator “tan”

Types for “tan”:

- `real`→`real`

Definition of the evaluation function $F_{tan}(T, V)$

- If V = `unknown` `then` $F_{tan}(T, V)$= `unknown`.
- Else if $F_{cos}(T, V)\neq 0$ then $F_{tan}(T, V)= F_{sin}(T, V)/ F_{cos}(T, V)$.
- Else $F_{tan}(T, V)$ is undefined and an attempt to evaluate it will set the *Exception* flag to `true`.

11.2.21. Operator “asin”

Types for “asin”:

- `real`→`real`

Definition of the evaluation function $F_{asin}(T, V)$

- If V = `unknown` `then` $F_{asin}(T, V)$= `unknown`.
- Else there exists a number X such that $F_{sin}(T, X)\neq V$ then $F_{asin}(T, V)= X$.
- Else $F_{asin}(T, V)$ is undefined and an attempt to evaluate it will set the *Exception* flag to `true`.

11.2.22. Operator “acos”

Types for “acos”:

- `real`→`real`

Definition of the evaluation function $F_{acos}(T, V)$

- If V = `unknown` `then` $F_{acos}(T, V)$= `unknown`.
- Else there exists a number X such that $F_{cos}(T, X)\neq V$ then $F_{acos}(T, V)= X$.

- Else $F_{acos}(T, V)$ is undefined and an attempt to evaluate it will set the *Exception* flag to `true`.

11.2.23. Operator "atan"

Types for "atan":

- `real`→`real`

Definition of the evaluation function $F_{atan}(T, V)$

- If $V =$ `unknown then` $F_{atan}(T, V) =$ `unknown`.
- Else there exists a number X such that $F_{tan}(T, X) \neq V$ then $F_{atan}(T, V) = X$.
- Else $F_{atan}(T, V)$ is undefined and an attempt to evaluate it will set the *Exception* flag to `true`.

11.2.24. Operator "random"

Types for "random":

- `real`

Definition of the evaluation function $F_{random}(T)$

- $F_{random}(T, V) = RandomNum$.

12. Loading Guidelines

The state of the guideline is initialised by the LoadGuideline operation, whose parameter is a text string *G* conforming to the syntax of <guideline> in the PRO*forma* BNF. The Proforma BNF defines a <guideline> by

<guideline> = [<directives>] <plan> {<task>|<data_item>}

So the guideline *G* is of the form $Dir\ RootDef_1 \ldots Def_N$, where:

- *Dir* is the directives block of the guideline, whose syntax is defined by <directives>,
- *Root* is the root plan, whose syntax is defined by <plan>, and
- $Def_1 \ldots Def_N$, are definitions of the data items and tasks in the plan (other than the root plan) whose syntax is defined by either <task> or <data_item>.

12.1. The Operation LoadGuideline(G)

Parameters:

- *G* is a text string having the syntax <guideline>

Definition:

1. Set *RandomNum* to a real value in the range 0 to 1 inclusive. We do not define how this value is chosen but the intention is that it should be generated in a pseudo-random manner.

2. Set the Exception flag to false and empty the Properties and Changes table.

3. Parse *G* as *Dir RootDef*$_1$ … *Def*$_n$, where *Dir* has the syntax <directives> *Root* has the syntax <plan>, and *Def*$_1$ … *Def*$_n$, are definitions of the data items and tasks in the plan (other than the root plan) whose syntax is defined by either <task> or <data_item>.

4. Select a "new" component identifier *C* ("new" meaning that *C* does not already occur in the Properties table).

5. Perform the operation InstantiateTask(*G*, *Root*, *C*)

6. For each *Def*$_i$ in the set of components *Def*$_1$ … *Def*$_N$, if *Def*$_i$ is a <data_item> then perform InstantiateDataItem(*Def*$_i$). If *Def*$_i$ is a <task> then ignore it (the instantiation of the root plan in step 2 will result in the instantiation of all the other tasks in the guideline).

12.2. The Operation InstantiateTask(G,T, C)

Parameters:

- *G* is a text string whose syntax is defined by <guideline> and which describes the guideline we are loading.
- *T* is text string whose syntax is defined by <task> and which describes the task we are instantiating. *T* will be part of the guideline *G*.
- *C* is a component identifier, which is to be used to identify the task *T*.

From the BNF for PRO*forma* set out in ¶3 it can be seen that *T* must be of the form *TaskClass* '::' *Name Att*$_1$ … *Att*$_N$ "end" *TaskClass* "." where

- *TaskClass* is one of "plan", "decision", "action", or "enquiry".
- *Name* is the name of the task, whose syntax is defined by <name>

- $Att_1 \ldots Att_N$ is the definition of task's attributes, whose syntax is defined by either {<plan_attribute>} {<decision_attribute>} {<action_attribute>} {<enquiry_attribute>} depending on the value of *TaskClass*.

As specified in **Definition 1** we call a text string of the above form a *Task Definition* for the task *Name.*

Definition:

1. Parse *T* as *TaskClass* '::' *Name* $Att_1 \ldots Att_N$ "end" *TaskClass* "." where
   a. *TaskClass* is one of "plan", "decision", "action", or "enquiry".
   b. *Name* is the name of the task, whose syntax is defined by <name>
   c. $Att_1 \ldots Att_N$ is the definition of task's attributes, whose syntax is defined by either {<plan_attribute>} {<decision_attribute>} {<action_attribute>} {<enquiry_attribute>} depending on the value of *TaskClass*.

2. Add (*C*,class,*ClassConst*) to the Properties table where *ClassConst* is either `plan`, `decision`, `action`, or `enquiry` depending on whether *TaskClass* is respectively "plan", "decision", "action", or "enquiry".

3. Add (*C*,state,`dormant`) to the Properties table.

4. For each Att_i in the attribute list $Att_1 \ldots Att_N$ perform the operation SetTaskAttribute(*C*, Att_i). This operation is defined in ¶12.5.

5. For each Att_i having the syntax of <component> perform the operation InstantiateComponent(*G*, *C*, Att_i). This operation is defined in ¶12.3. Note that only a plan has attributes of the form <component>.

12.3. The Operation InstantiateComponent(G, C_T , A).

Parameters:

- A text string *G* whose syntax is defined by <guideline> and which describes the guideline we are loading.
- A component identifier C_T.
- A text string *A* whose syntax is defined by <component >.

Definition:

1. Parse *A* as "component" "::" *Name* $Att_1 \ldots Att_n$ where *Name* has the syntax <task_name> and each Att_i $(1 \le i \le n)$ has the syntax <component_attribute>.

2. Find the Task Identifier C_A such that (C_A,name,*Name*) and (C_A,parent_plan,C_T) are both in the Properties table. These rows will have been created by the SetTaskAttribute Operation (¶12.5).

3. If the Properties table contains the row (C_A,`class`,`decision`) or (C_A,`class`,`action`) then add the row (C_A,`confirmatory`,`true`) to the Properties table otherwise add the row (C_A,`confirmatory`,`false`). Note that this establishes a default value for the `confirmatory` property which may be overridden by the SetComponentAttribute operation.

4. For each Att_i $(1 \le i \le N)$ perform the operation SetComponentAttribute(C_A,C_T,Att_i)

5. Let T_A be the task definition for task *Name* in guideline *G*. The meaning of "Task Definition" is given in Definition 1, the context sensitive syntax of PRO*forma* guarantees that *G* will contain exactly one task definition for task *Name.*

6. Perform the operation InstantiateTask(*G*, T_A, C_A).

12.4. The Operation SetComponentAttribute(C,C_P,A)

Parameters:

- *C* is a task Identifier.
- C_P is a task identifier, which identifies the parent task of *C*.
- *A* is a text string of the form <component_attribute>.

Definition:

1. If *A* is of the form ""schedule_constraint" "::" "completed" "(" *Name* ")" ";" then
   - Find the task identifier C_A such that the Properties Table contains the rows (C_A , name, *Name*) and (C_A , parent_plan, C_P). These rows will have been created by the SetTaskAttribute Operation (¶12.5).
   - If the Properties table contains a row (*C*,antecedent_tasks,<T_1,…,T_N>) where <T_1,…,T_N> is a sequence of task identifiers then replace this row with (*C*,antecedent_tasks,<T_1,…,T_N ,C_A>).
   - Otherwise add (*C*,antecedent_tasks,<C_A>) to the Properties table where <C_A> is the sequence containing just C_A.

2. If *A* is of the form "param_value" "::" *Name = E* ";" where *Name* is an <*atom*> and *E* is an <expression> then
   - Create a new component identifier *P*
   - Add the rows (*P*, class, `parameter`), (*P*,expression,*E*) and (*P*,name,*Name*) to the Properties table.
   - If the Properties table contains a row (*C*,parameters,<P_1,…,P_N>) where <P_1,…,P_N> is a sequence of text strings then replace this row with (*C*,parameters,<P_1,…,P_N ,*P*>).
   - Otherwise add (*C*,parameter_assignments,<*P*>) to the Properties table where <*P*> is the sequence containing just *P*.

3. If *A* is of the form "cycle_repeat" "::" *E U* where *E* is an <expression> and *U* is a <time_unit> then and add (*C*,cycle_interval,*E*) (*C*,cycle_interval_unit ,*U*) to the Properties table.

4. If *A* is of the form "autonomous" "::" "yes" resp. "autonomous" "::" "no" **and** the Properties table does not contain the row (C_A,`class`,`enquiry`) or (C_A,`class`,`plan`) then add the row (C_A,`confirmatory`,`true`) resp. (C_A,`confirmatory`,`false`) to the Properties table.

   Note that the "autonomous" keyword is ignored for enquiries and plans.

5. Otherwise add the row indicated by Table 1 below.

Form Of *A*	**Row Added To The Properties Table**
"optional" "::" "yes" ";"	(*C*,optional,`true`)
"optional" "::" "no" ";"	(*C*,optional,`false`)
"terminal" "::" "yes" ";"	(*C*,terminal,`true`)
"terminal" "::" "no" ";"	(*C*,terminal,`false`)
"ltwh" "::" N_1 "," N_2 "," N_3 "," N_4 ";" (each N_i having the syntax <integer>)	(*C*,lwth,<V_1,V_2,V_3,V_4>) where V_1,V_2,V_3,V_4 are the numeric values of the integers N_1,N_2,N_3,N_4.
"number_of_cycles" "::" *N* ";" (*N* having the syntax <integer>)	(*C*,lwth,*V*) where *V* is the numeric value of *N*.

"cycle_until" "::" *E*	(*C*,cycle_until,*E*)

Table 1 Rows Added By SetComponentAttribute

12.5. The Operation SetTaskAttribute(G,C,Att)

Parameters:

- *C* is a task identifier that has previously been chosen for this task. In the case of the root plan this identifier will have been chosen by the LoadGuideline operation, for any other task it will have been chosen as a result of performing InstantiateTask (¶12.2) on its parent plan.
- *Att* is a text string describing the attribute value to be set. The syntax of *Att* is defined by either <generic_attribute> <plan_attribute>, <decision_attribute>, <action_attribute>, or <enquiry_attribute>[3].

Definition:

1. If *Att* is of the form "component" "::" *Name CompAtts* where *Name* has the syntax <task> and *CompAtts* has the syntax {<component_attribute>} then
    a. Create a new Task Identifier *T*
    b. Add the row (*T*,name,*Name*) to the Properties table.
    c. Add (*T*,parent_plan,*C*) to the Properties table.

2. Else if *Att* begins with the reserved word "candidate" then perform the operation InstantiateCandidate(*C*,*Att*)

3. Else if *Att* begins with the reserved word "source" then perform the operation InstantiateSource(*C*,*Att*)

4. Else If *Att* is of the form "parameter" "::" P_1 ","…"," P_N ";" then:

    a. Parse each P_i as

    Name ["attributes" ["type" "::" *Type*] [<generic_attribute_list>]]

    where square brackets denote optional components, *Name* is a <parameter_name> and *Type* , if present is a <data_type>

    b. Look for a component identifier C_P such that the Properties table contains the rows (C_P,name,*Name*), (C_P,class,`parameter`). If no such identifier is found then create it and add those two rows.

    c. If Type is present then (C_P,type,*Type*) to the Properties table otherwise add (C_P,type,`text`) to the Properties table, where C_P is the component identifier found or created in step b.

[3] Or by all four of these if the attribute is a <generic_task_attribute> .

d. If a <generic_attribute_list> is present and defines a caption *Cap* then add (C_P,caption,`Cap`) to the Properties table, where C_P is the component identifier found or created in step b.

e. If the <generic_attribute_list> is present and defines a description *Desc* then add (C_P,description,`Desc`) to the Properties table, where C_P is the component identifier found or created in step b.

5. Else add to the Properties table the rows specified in Table 2 below.

Form Of *Att*	**Row Added To The Properties Table**
"caption" "::" *Cap* ";"	(*C*,caption,*Cap*)
"description" "::" *D* ";"	(*C*,definition,*D*)
"goal" "::" *G* ";"	(*C*,goal,*G*)
"precondition" *P* "::""";"	(*C*,precondition,*P*)
"wait_condition" *P* "::""";"	(*C*,wait_condition,*P*)
"trigger" "::" *T* ";"	(*C*,trigger,*T*)
"postcondition" "::" *P* ";"	(*C*,postcondition,*P*)
"abort" "::" *A* ";"	(*C*,abort_condition,*A*)
"terminate" "::" *T* ";"	(*C*,terminate_condition,*T*)
"choice_mode" "::" ["single"|"multiple"] ";"	(*C*,choice_mode,*M*) where *M* is either the constants `single` or the constant `multiple` as appropriate.
<support_mode> = "support_mode" "::" ["symbolic" | "numeric"] ";"	(*C*,support_mode,*M*) where *M* is either the constant `symbolic` or the constant `numeric` as appropriate.
"procedure" "::" *P* ";"	(*C*,procedure,*P*)
<context> = "context" "::" *Con* ";"	(*C*,context,*Con*)

Table 2 Rows Added Properties Table By SetTaskAttribute

12.6. The Operation InstantiateCandidate(D,Cand)

Parameters:

- *D* is a component identifier identifying a decision.
- *Cand* is a text string whose syntax is defined by <candidate> and which describes a candidate of the decision *D*.

Definition:

1. Parse *Cand* as "candidate" "::" *Name* ";" $A_1 \ldots A_N$ where *Name* is a <candidate_name> and each A_i is a <candidate_attribute>.

2. Create a new component identifier *C*.

3. Add (*C*, class, `candidate`) to the Properties table.

4. Add (*C*, name, *Name*) to the Properties table.

5. If the Properties table contains a row (*D,* candidates, <C_1, …,C_N>) then replace this row with (*D*, candidates, <C_1,…,C_N, *C*>) otherwise add (*D*,candidates,<*C*>) to the Properties table.

6. For each A_i (where $1 \le i \le n$):
    a. If A_i is an <argument> then perform the operation InstantiateArgument(*C*,A_i) as defined in ¶12.7.
    b. Otherwise add a row to the Properties table as set out in below.

Form Of A_i	Row Added To The Properties Table
"caption" "::" *Cap* ";"	(*C*,caption,*Cap*)
"description" :: *D* ";"	(*C*,description,*D*)
"rule" "::" *E* ";"	None
"recommendation" "::" *E* ";"	(Cand, recommendation, *E*)
<priority> = "priority" "::" *N* ";" where *N* is an <integer>	(Cand, priority, *V*) where *V* is the numeric value of *N*.

Table 3 Rows Added by the InstantiateCandidate Operation

12.7. The Operation InstantiateArgument(C,A)

Parameters:

- *C* is a component identifer identifying a candidate.
- *A* is a text string with the syntax of <argument>, which defines an argument of the candidate *C*.

Definition:

1. Parse *A* as "argument" "::" *S* "," *E* , ["attributes" ["name" "::" *Name* ";"] [<generic_attribute_list>]]";" where square brackets denote optional components, *S* is a <support>, *E* is an <expression>, and *Name*, if present is an <argument_name>.

2. Create a new component identifier C_A.

3. Add (C_A, class, `argument`) to the Properties table.

4. If the properties table contains a row (*C*,arguments, <C_1, …, C_N>) then replace this row with (*C*,arguments, <C_1, …, C_N, C_A>), otherwise add (*C*,arguments,<C_A>) to the properties table.

5. Add (C_A, support, V) to the properties table where V is `for`, `against`, `confirming`, `excluding` or the numeric value of S depending on whether S is "for", "against", "confirming", "excluding" or an <integer>.

6. If *Name* was present then add (C_A, name, *Name*) to the Properties Table.

7. If the generic attribute list defines a caption *Cap* then add (C_A, caption, *Cap*) to the Properties Table.

8. If If the generic attribute list defines a description *desc* then add (C_A, description, *desc*) to the Properties Table.

12.8. The Operation InstantiateSource(C,S)

Parameters:

- C is a task identifier and identifies a decision or enquiry.
- S is a text string whose syntax is defined by <source>.

Definition:

1. Parse S as "source" "::" *Name* ";" $A_1 \ldots A_N$ where *Name* is a <data_name> and $A_1 \ldots A_N$ are <source_attribute>s

2. Create a new component identifier C_S.

3. Add (C_S, class, `source`) to the Properties table.

4. Add (C_S, name, *Name*) to the Properties table.

5. If the Properties table contains a row (C, sources, $<C_1, \ldots, C_N>$) then replace this row with (C, sources, $<C_1, \ldots, C_N, C_S>$) otherwise add (C, sources, $<C_S>$) to the Properties table.

6. If there is no component identifier D such that the Properties table contains the rows (D,name,*Name*) and (D, class, `data_item`) then create a new identifier D and add those two rows to the Properties table.

7. Add the row (C_S, data_item,D) to the Properties table where D is the component identifier created or found in step 5 above.

8. For each A_i ($1 \leq i \leq N$) add to the Properties table the row specified by Table 4 below.

Form Of A_i	Row Added To The Properties Table

“caption” “::” *Cap* “;”	(*C*,caption,*Cap*)
“description” “::” *D* “;”	(*C*,description,*D*)
“mandatory” “::” “yes” “;”	(*C*,mandatory,`true`)
“mandatory” “::” “no” “;”	(*C*,mandatory,`false`)

Table 4 Rows Added By InstantiateSource

12.9. The Operation InstantiateDataItem(Def).

Parameters:

- *Def* is a text string with the syntax

Definition:

1. Parse *Def* as “data” “::” *Name* “type” “::” *Type* “;”A_1 … A_N “end” “data” “;” where *Name* is a <data_name>, *Type* is a <data_type> and each A_i is a <data_attribute> (for $1 \le i \le N$).

2. Create a new component identifier *C*.

3. Add the row (*C*, class, `data_item`) to the Properties table.

4. Add (*C*, name,*Name*) and (*C*, type, *Type*) to the Properties table.

5. For each each A_i ($1 \le i \le N$) :

    a. If A_i is of the form “warning_condition” “::” *K* “,” *E* “;” where *K* is a <constant> and *E* is an <expression> then create a new component identifier *W* and add the rows (*W*,name,*K*) and (*W*,condition,*E*) to the Properties table.

    b. Else add to the Properties table the row specified by Table 5 below.

Form Of A_i	**Row Added To The Properties Table**
“caption” “::” *Cap* “;”	(*C*,caption,*Cap*)
“description” “::” *D* “;”	(*C*,description,*D*)
“range” “::” $T_1, \ldots, T_n$ “;”for each *i* such that $1 \le i \le n$, T_i is a <textual_constant>.	(*C*,range, $\langle V_1, \ldots, V_n \rangle$) where, for each *i* such that $1 \le i \le n$, V_i is T_i with any enclosing single or double quotes removed.
“range” “::” $N_1, \ldots, N_n$ “;”for each *i* such that $1 \le i \le n$, N_i is a <number>.	(*C*,range, $\langle V_1, \ldots, V_n \rangle$) where, for each *i* such that $1 \le i \le n$, V_i is the numeric value of N_i .

“default_value” “::” *C* “;” where *C* is a <constant>.	(*C*,default_value, *V*) where *V* is the value of *C*.
“true_value” “::” *T* “;”	(*C*,true_value, *T*)
“false_value” “::” *F* “;”	(*C*,false_value, *F*)
“mandatory_validation” “::” *E* “;”	(*C*,mandatory_validation, *E*)
“derivation” “::” *E* “;”	(*C*,derivation, *E*)
“unit” “::” *U* “;”	(*C*,unit, *U*)

Table 5 Rows Added By InstantiateDataItem