



BART

-

USER MANUAL

Réf. :	Version : 1.0	Date :
--------	---------------	--------

CLEARSY Société par Actions Simplifiée au Capital de 266 880 Euros. - RCS Aix-en-Provence 433 901
402 - Code NAF 721 Z
320, Avenue Archimède – Les Pléiades III - Bât A - 13857 AIX EN PROVENCE CEDEX 3
Tél : 04 42 37 12 70 – Fax : 04 42 37 12 71

REVISIONS

Version	Date	Comment
1.0		Initial version

INDEX

Revisions.....	2
Index.....	3
I Introduction.....	6
II Usage.....	7
II.1 Command usage.....	7
II.2 Input files.....	7
II.3 Visibility for loaded components.....	8
II.4 Bart standard output verbosity.....	8
II.5 Bart rule trace.....	9
III Automatic refinement principles.....	10
III.1 Refined elements.....	10
III.1.1 Abstract variables.....	10
III.1.2 Operations.....	10
III.1.3 Initialisation.....	11
III.1.4 Process.....	11
III.2 Pattern-Matching.....	11
III.2.1 Jokers syntax.....	12
III.2.2 Pattern matching.....	12
III.3 Refinement rules.....	13
III.3.1 Introduction.....	13
III.3.2 Constraints.....	13
III.3.3 Guards.....	14
III.3.4 Rule checking process.....	15
III.3.5 Jokers use in result.....	15
III.4 Hypothesis stack – Environment analysis.....	16
III.5 Result production and writing.....	17
IV Bart guards – Predicate synonyms.....	18
IV.1 Guards.....	18
IV.1.1 Expression guards.....	18
IV.1.2 Predicate guards.....	19
IV.1.3 Substitution guards.....	20
IV.2 Predicate synonyms.....	20
V Pragmas and comments.....	22
V.1 EMPILE_PRE, DEPILE_PRE.....	22
V.2 Magic.....	23
V.2.1 For variables.....	23
V.2.2 For substitutions.....	23
V.3 CAND.....	23
VI Rule files.....	25
VI.1 Syntax.....	25

VI.2 Using rule files.....	26
VI.2.1 Providing rule files on command line.....	26
VI.2.2 Rule file associated to the component.....	26
VI.2.3 Bart refinement rule base.....	26
VII Variables refinement.....	27
VII.1 Variable theories syntax.....	27
VII.2 Variable rule research.....	28
VII.3 Storing information predicates about found variable rules.....	30
VII.4 Invariant for refined abstract variables.....	30
VII.5 Specifying variable refinement results.....	31
VII.5.1 Using CONCRETE VARIABLES clause.....	31
VII.5.2 Using REFINEMENT VARIABLES clause.....	32
VIII Substitution refinement.....	34
VIII.1 Rule syntax.....	34
VIII.2 Rule research.....	35
VIII.3 Refinement process.....	36
VIII.4 Default refinement behaviours.....	40
VIII.5 Special refinement substitutions.....	42
VIII.5.1 Iterators.....	42
VIII.5.2 Using operations from seen machines - SEEN OPERATION.....	48
VIII.5.3 Defining imported operations - IMPORTED OPERATION.....	49
VIII.5.4 Controlling the refinement process.....	52
VIII.5.5 Local variable declarations.....	54
VIII.6 Declaring operation refinement variables.....	55
VIII.7 Usage of substitution rules.....	55
VIII.7.1 Structural and operation rules - Operation refinement.....	56
VIII.7.2 Initialisation rules.....	59
IX Tactic and user pass theories.....	62
IX.1 User pass theory.....	62
IX.1.1 Syntax.....	62
IX.1.2 Usage.....	62
IX.2 Tactic theory.....	63
IX.2.1 Syntax.....	63
IX.2.2 Usage.....	63
IX.3 Priority of Tactic and User pass theories.....	64
X Result production and writing.....	66
X.1 Formatting the result.....	66
X.2 Implementing results.....	67
X.2.1 Splitting operations in output components.....	67
X.2.2 Resolving deadlocks.....	68
XI Appendix A – Figures table.....	72
XII Appendix B – Syntax elements table.....	74
XIII Appendix C – Rule files complete syntax.....	75
XIII.1 Rule files.....	75
XIII.2 Variables refinement rules.....	75
XIII.3 Initialisation refinement rules.....	76
XIII.4 Operation refinement rules.....	76

XIII.5 Structural refinement rules.....	77
XIII.6 User pass theory.....	77
XIII.7 Tactic theory.....	77
XIII.8 Predicate synonyms theory.....	78
XIII.9 Substitutions.....	78
XIII.10 Predicates.....	80
XIII.11 Expressions.....	81
XIII.12 Diverse.....	82

I INTRODUCTION

BART is a tool that can be used to automatically refine B components. This process is rule based so that the user can drive refinement. Its own rule language has been defined in this purpose.

It has been designed to be a stand-alone tool, but it may be launched from AtelierB user interface.

II USAGE

II.1 Command usage

The Bart command syntax is as follow (help message displayed by command launching without parameters):

```
Bart { -r rule_file } -m machine_file
```

Options:

Parameter	Comment
-h	Displays this help message
-d	Debug. This forces Bart to display back all the loaded data
-I dir	Adds the given directory to the list of directories searched for machine files
-v	Displays more information
-V	Displays more information than -v
-s machine_name	Adds a seen machine
-o operation_name	Only refine the given operation
-a file_name	Visibility file
-e	Handles duplicate names in rmf files as error instead of warning
-p project	Name of the project that should be loaded (requires -b)
-b path	Path to the bdp of the project
-H file	Indicates a file containing the header that should be inserted in the generated machines
-t	Writes rule trace in the result
-g file	Writes the list of generated files to file
-D dir	Writes the generated files to the given directory
-x	Displays output as Xml
-X file	Writes input machine as Xml
-l	Displays guards list
-f name	Use given resolving information for finding path of given component file

Figure 1 : Bart command line parameters

II.2 Input files

As an input, Bart must be given at least the machine or refinement (.mch or .ref file) to refine. This file path must be given to Bart using -m parameter. This given file path can be relative or absolute. There must be exactly one component to refine.

Furthermore, user may provide rule files to process refinement of given component. These files are .rmf suffixed, and are given using -r parameter. User can provide zero, one or more rule files. Their path can be relative or absolute.

II.3 Visibility for loaded components

When Bart must load seen machines or given component abstraction, it must be able to find their associated files on the file system. So at the command launching user must provide necessary information. There are three ways to do this:

- *-I dir*: This option allows the user to directly specify directories components to load must be searched in. So there can be several *-I* parameters on command line.
- *-a file_name*: This is used to give Bart a visibility file. Each line of this file is a research directory. This option could be used together with *-I* option, in this case file directories and command line directories are added
- *-b path and -p project*: With these options, information about an AtelierB project is provided for searching components. *-p* option indicates the project name, and *-b* is the project bdp path. *-b* and *-p* must be present together.

All these options can not be used at the same time. Only AtelierB project resolving is used if all these parameters are given on command line.

II.4 Bart standard output verbosity

In standard output mode, Bart prints result of variables, operations and initialisation refinement on standard output.

Variable refinement result is the list of found rules associated to their variables. In standard output mode, operation and initialisation refinement result is symbolized with "+" (rule found) and "-" (no rule could be found) characters. For example:

```
Refining operation operation_test
++++++
Refinement of operation_test finished
```

Figure 2 : Example of Bart standard output

On the command line, the detail level of output can be increased with *-v* (verbose mode) and *-V* (very verbose mode) options.

In verbose mode, the output for previous operation refinement would be as follow:

```
Refining operation operation_test
Rule found: theory1.rule1
Rule found: theory1.rule2
Rule found: theory1.rule3
Rule found: theory2.rule1
Rule found: theory2.rule2
Rule found: theory2.rule3
Refinement of operation_test finished
```

Figure 3 : Example of Bart verbose output mode

The failure character “-” is replaced by a “No rule could be found” message when launching Bart in verbose mode.

II.5 Bart rule trace

There are two ways to keep a trace of rules applied by Bart.

Each time a component is refined with Bart, the tool generates a file with same name as the component with a .rs extension (example: machine.rs for machine.mch or machine_r.ref). This file contains name of the rules used to refine each element.

Furthermore, user may add -t parameter on command line. This option indicates to Bart that it must write used rule names in comments in generated components.

III AUTOMATIC REFINEMENT PRINCIPLES

Automatic refinement is a rule based refinement process for B components (abstractions or refinements). The tool is given a component, and it searches, for each element to refine, some rules that specify how it must be treated.

This section describes basic principles of automatic refinement.

III.1 Refined elements

III.1.1 Abstract variables

First elements treated by Bart tool are abstract variables of component to refine (content of the ABSTRACT_VARIABLE clause). The tool must produce, for each one of them, one or more abstract or concrete variables that implement it.

III.1.2 Operations

Bart processes operations of given component in order to refine them. It must produce, for each operation, a substitution body concrete enough to be put in the component implementation.

Refined operations are considered for the whole component abstraction. It means that Bart refines most concrete version of each operation. Here is an example of this process:

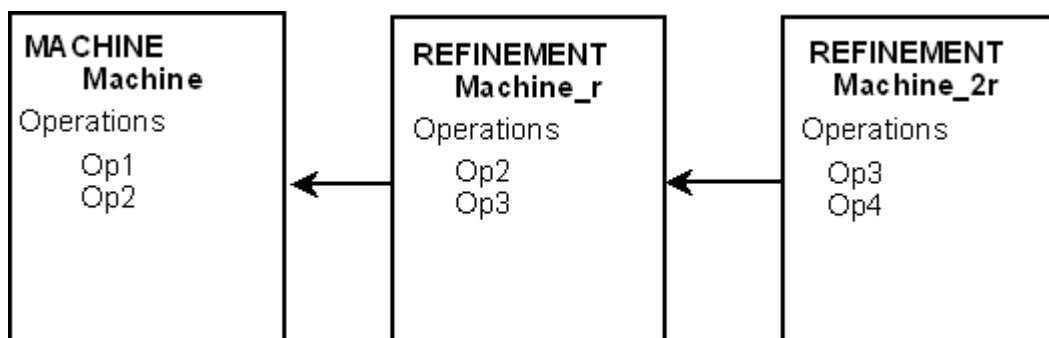


Figure 4 : Example of selection of operations to refine

For this example, if the given component to refine is Machine_2r, operations processed by Bart will be:

- *Op1* from *Machine*
- *Op2* from *Machine_r*
- *Op3* from *Machine_2r*

- *Op4* from *Machine_2r*

III.1.3 Initialisation

Bart also refines content of initialisation clause of given component. Typically, it produces a concrete result by specifying initialisation substitutions for concrete variables refining content of ABSTRACT_VARIABLES clause.

III.1.4 Process

The following draw presents the order of previously described refinement steps.

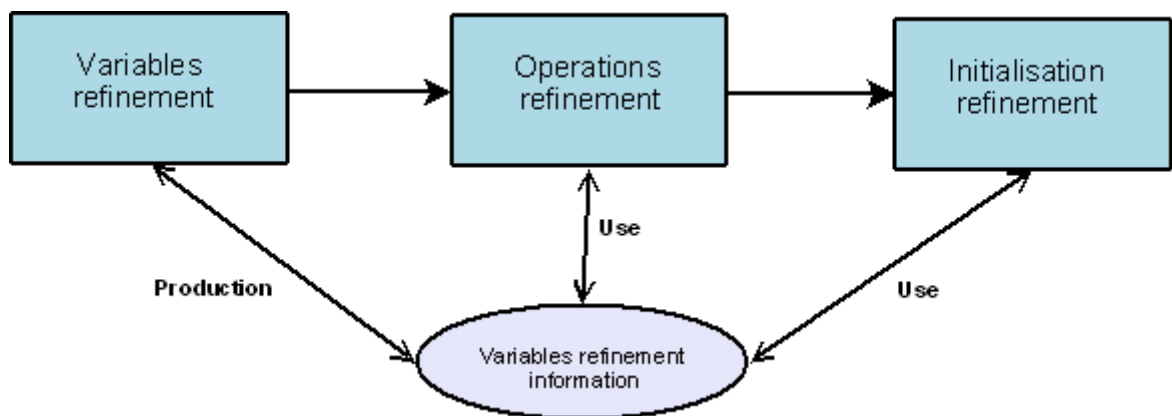


Figure 5 : Refinement process order

Abstract variables are refined first, as other parts of the process need its output to find suitable rules for operations and initialisation. It is necessary at these steps to know how variables have been refined.

This variable information is stored as predicates in Bart hypothesis stack (cf. III.4).

As it will be described later, refinement process uses rules to determine how each element is refined. A same rule can apply for several elements, so it must be general. In this purpose, the rule language uses *jokers*, so that rules can contain variable parts.

III.2 Pattern-Matching

A large part of the refinement process uses the concept of pattern-matching. In Bart rule language, user can define patterns, containing jokers, which will be matched against real B elements.

III.2.1 Jokers syntax

In Bart, jokers are '@' character followed by a single letter. For example, @a, @x and @t are valid Bart jokers.

@_ is a special joker, used for special treatments in pattern-matching.

III.2.2 Pattern matching

Bart jokers can be used to write general expression, predicate or substitution patterns. These patterns can be matched against B elements.

Each pattern-matching action has a result status, as it may be either a success or a failure, and instantiates jokers that it contains.

A simple joker matches with any B element. A complex pattern matches a B element if each of its contained jokers can be instantiated with a subpart of the element. If a joker appears several times in a pattern, it has the same value in a unique instantiation

If the pattern-matching is a success, jokers contain element subparts that made the match successful. The @_ joker is a special one, as it means that its instantiation has not to be stored. So @_ joker can stand for different elements in a same pattern if it appears several times.

The following table shows examples of successful or failed pattern-matching, with their status and associated jokers instantiation.

Pattern	Element	Status	Jokers instantiation
@a	aa	Success	{@a = aa}
@a	aa + bb	Success	{@a = aa + bb}
@a + @c	yy + 2	Success	{@a = yy, @c = 2}
@a + @c	yy - 2	Failure	-
@a + @c	(aa + 1) + f(3)	Success	{@a = aa + 1, @b = f(3)}
@a + @b * @a	aa + bb * 2	Failure	-
@a + @b * @a	aa + bb * aa	Success	{@a = aa, @b = bb}
not(@p)	not(vv < 0)	Success	{@p = vv < 0}
@a	IF val THEN aa := 0 ELSE aa := 1 END	Success	{@a = IF val THEN aa := 0 ELSE aa := 1 END }
IF @p THEN @t ELSE @e END	IF val THEN aa := 0 ELSE aa := 1 END	Success	{@p = val, @t = aa :=0, @e = aa := 1}
IF @_ THEN @t ELSE @e END	IF val THEN aa := 0 ELSE aa := 1 END	Success	{@t = aa :=0, @e = aa := 1}

Figure 6 : Examples of pattern-matching without previous instantiation

In some cases, some jokers may already be instantiated when the pattern matching is done. An instantiated joker matches an element if its stored value is equal to the element.

For example, if pattern is $@a + @b$, following table shows how pattern-matching is done if some jokers are already instantiated.

Element	Original instantiation	Status	Result instantiation
$1 + 3$	$\{ @a = 2 \}$	Failure	-
$1 + 3$	$\{ @a = 1 \}$	Success	$\{ @a = 1, @b = 3 \}$
$aa + (1 + bb)$	$\{ @b = bb \}$	Failure	-
$aa + (1 + bb)$	$\{ @b = 1 + bb \}$	Success	$\{ @b = 1 + bb, @a = aa \}$
$var1 + (var2 - 1)$	$\{ @a = var1, @b = var2 - 1 \}$	Success	$\{ @a = var1, @b = var2 - 1 \}$

Figure 7 : Examples of $@a + @b$ pattern-matching with previous instantiation

III.3 Refinement rules

III.3.1 Introduction

Bart uses rules for refining variables, operations and substitutions. These rules belong to different types: variables rules, or substitution rules, which can be used for both operations and initialisation. Rules of same type are gathered in theories.

Rules usually contain a pattern, and may contain a constraint. These two elements are used to know if a rule can be applied to refine a certain element. Rules also contain clauses that express the refinement result.

III.3.2 Constraints

Rules may have constraints, expressed in their WHEN clause. A constraint is a predicate, which may contain jokers. It may be a complex predicate, built with "&" and "or" operators.

Bart contains a stack of hypothesis (cf. III.4), which is built from the machine to refine and its environment. A constraint is successfully checked if its elementary elements (element not containing "&" or "or") can be pattern-matched with a predicate of the stack so that the complex constraint is true. According to operators, Bart uses backtracking to try every combination of instantiation that should be a success.

If several instantiations can make the constraint be successfully checked, Bart uses one of them. In this case, it is better to write a more detailed constraint to have only one result. If there are several results, Bart could choose one which is not what the user had planned.

Usually, when checking a constraint, some jokers have already been instantiated.

Here are some examples of constraint checking, if the hypothesis stack contains the following predicates:

```
Stack =
  {bb <= 0,
   var = bb,
   mm : INT,
   nn = 2,
   nn : INT}
```

Figure 8 : Hypothesis stack for constraint checking examples

Constraint	Original instantiation	Status	Result instantiation
@a <= 0	{}	Success	{@a = bb}
@a <= 0	{@a = cc}	Failure	-
@a <= 0 & (@b = 0 OR @b = @a)	{@a = bb}	Success	{@a = bb, @b = var}
@a : INT & @a = 2	{}	Success	{@a = nn} (Bart tries mm but it fails, so the joker is instantiated with nn)

Figure 9 : Examples of constraint checking

III.3.3 Guards

Guards are special predicates which may be present in rule constraint clauses. They allow checking some properties on elements to refine and their environment.

There are two kinds of guards: some are simply present in the predicate stack. They are added at the environment loading. For instance ABCON (abstract constant), ABVAR (abstract variable) belong to this kind of guards.

The other kind is calculated guards. For these ones, during constraint checking, Bart doesn't try to match them with the stack, but directly calculates if the guard is true or false. This kind of guards may also have side effects. For example bnum (numeric test) or bident (identifier test) are calculating guards.

Guards are simply put in the constraint as regular predicates.

Example: @a <= @b & ABVAR(@b) & bnum(@a), with @b instantiated.

It checks that @b is an abstract variable (predicate present in the stack), that a predicate @a <= @b is present, and that @a from this predicate is a numeric value (checked by the tool with computations).

III.3.4 Rule checking process

The following figure presents how Bart determines if a rule can be used to refine an element:

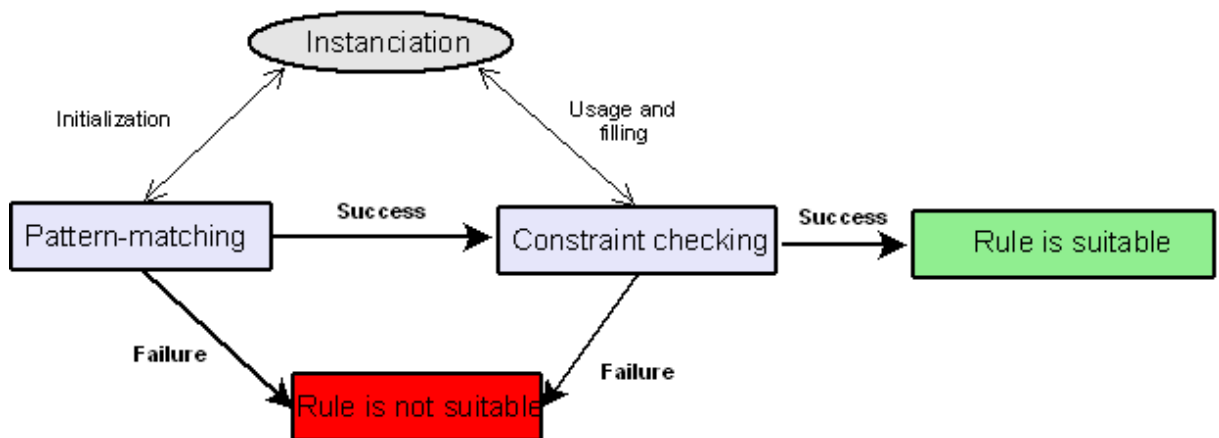


Figure 10 : Bart testing rule process

This process is used for variables, operations and initialisation refinement, although it is simpler for variables.

Every rule contains a pattern. First Bart tries to match it with the element to refine. If it succeeds, it tries, if the rule has a constraint clause, to check it against hypothesis. When checking the constraint, some jokers have already been instantiated by pattern matching. If the constraint checking is a success or the rule had no constraint, then it will be used to refine current element.

Variable process is simpler as variable rules have simple pattern, which is a single joker (cf. IV). Variable rule patterns are only matched in order to instantiate the joker representing currently refined abstract variable. This joker is reused in WHEN or result clauses.

III.3.5 Jokers use in result

Once a rule has been chosen to refine an element, Bart must build refinement results. These results are specified in dedicated clauses of variable or substitution rules.

Jokers that have been instantiated by the rule selecting process are reused in the result specification. Those which have been instantiated with identifiers

can be reused to build new identifiers. For instance, if @i joker was present in pattern and its value is "ident", user can provide @i_r value in the result. This value will be "ident_r" after instantiation.

For substitution rules, result pattern is a substitution. For variable rules, it is a list of refinement variables identifiers, invariant and initialisation. For building the result, Bart replaces in this pattern all joker occurrences with their values previously calculated.

III.4 Hypothesis stack – Environment analysis

At launch, Bart builds an hypothesis stack with predicates coming from the machine to refine and its environment, and with guards, which are predicates giving more information about environment.

This section shows which parts of the environment are analysed to fill the predicate stack.

Machine	Predicate stack
All machines (component to refine, abstraction, seen machines)	
INVARIANT I	I &
ASSERTIONS A	A &
Seen Machines only	
PROPERTIES P	P &
SETS S; E = {v1, ..., vn}	SET(S) & ENUM(E) & v1 : E &...& vn : E & COCON(v1) & & COCON(vn) &
CONCRETE_CONSTANTS CC1, CC2	COCON(CC1) & COCON(CC2) &
CONCRETE_VARIABLES CV1, CV2	COVAR(CV1) & COVAR(CV2) &
ABSTRACT_VARIABLES AV1, AV2	ABVAR(AV1) & ABVAR(AV2) &
ABSTRACT_CONSTANTS AC1, AC2	ABCON(AC1) & ABCON(AC2) &
OPERATIONS par1 ← op1(par2) = body1 ; op3(par3) = body2	DECL_OPERATION(par1 ← op1(par2) body1) & DECL_OPERATION(op3(par3) body2)
Component to refine only	
ABSTRACT_VARIABLES AV3, AV4	REFVAR(AV3) & REFVAR(AV4)

Figure 11 : Hypothesis stack filling with environment

This table presents only how parts of given component environment are used to fill the stack. Bart doesn't necessarily add predicates in this exact order.

Some others stack guards will be added to the stack during refinement process. These guards will be only presented in IV, as they are not a part of the

initial environment analysis. Variables refinement also adds type predicates to the stack (cf. VII.3).

III.5 Result production and writing

Once every element (variables, operations and initialisation) has been refined, Bart must write the result. For a unique component, there may be several output components.

Operation refinement process may define new operations called in original ones refinement results. Furthermore, sometimes some operations can't be implemented in the same component. So Bart output is actually a chain of output components, each implementation importing the following machine. Original variables and operations, and new operations, are implemented along the chain.

For instance, following figure shows what could be a Bart output, when refining the component "Machine":

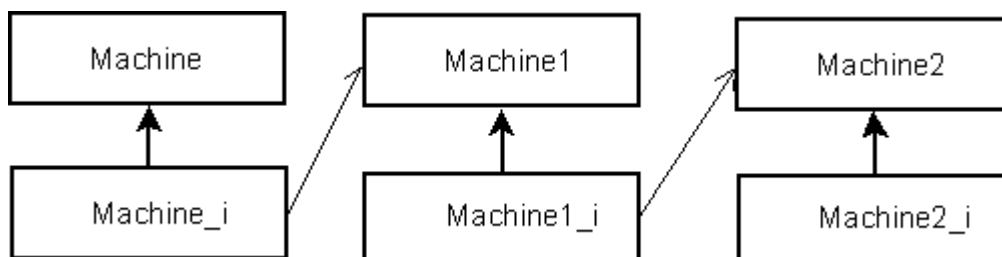


Figure 12 : Example of Bart output components

Thinnest arrows are importation links, and thick ones are refinement links.

If an operation refinement result calls a new imported operation, the new one must be defined and implemented further in the chain.

IV BART GUARDS – PREDICATE SYNONYMS

IV.1 Guards

Following tables describe Bart predefined guards, with their name, type (stack guard or calculated guard) and short descriptions of their meaning and side effects.

Guards description is available on command-line using the `-l` parameter with bart executable. Users can add new guards, by adding suitable classes to Bart library. Using the command line will display all registered guards, so it may print more information than this section.

Calculated guards usually must have all their joker instantiated to be used, except if the description explicitly says not. Most of stack guards should have their joker instantiated, although it is not mandatory.

For example, user could write a simple constraint as `ABVAR(@a)` where `@a` joker is not instantiated by rule pattern matching. This means the constraint checks if at least one abstract variable is present in seen machines, and `@a` is instantiated with one of seen machines abstract variables identifiers, if any.

IV.1.1 Expression guards

Guard	Type	Description
ABCON(expr)	Stack	Checks if given expression is an identifier of a seen machine abstract constant
ABVAR(expr)	Stack	Checks if given expression is an identifier of a seen machine abstract variable
B0EXPR(expr)	Calculated	Checks if given expression is a B0 expression
bident(expr)	Calculated	Checks if given parameter is an identifier
bnum(expr)	Calculated	Checks if given expression is a numeric literal
bpattern(expr1,expr2)	Calculated	Tries to make expr2 match with expr1. expr2 may be not fully instantiated If the match is successful, jokers of expr2 are instantiated
COCON(expr)	Stack	Checks if given expression is an identifier of a seen machine concrete constant
COVAR(expr)	Stack	Checks if given expression is an identifier of a seen machine concrete variable
ENUM(expr)	Stack	Checks if given expression is an enumerated set identifier from a seen machine
match(joker,expr)	Calculated	"joker" must be a single joker. This guard makes

		joker match with expr. Joker may be uninstantiated. If joker is not instantiated, the guard is true and joker value is now expr If joker is instantiated, guard is true if g can match with joker instantiation.
PAR_IN(expr)	Stack	Checks if given expression is an identifier of a currently refined operation input parameter. These guards are added to the stack when a new operation refinement begins
PAR_OUT(expr)	Stack	Checks if given expression is an identifier of a currently refined operation output parameter. These guards are added to the stack when a new operation refinement begins
REFINED(expr)	Stack	Checks if given expression is an identifier of a variable introduced by another variable refinement. These guards are added after variables refinement phase
REFVAR(expr)	Stack	Checks if given expression is an abstract variable of the component to refine
SET(expr)	Stack	Checks if parameter is a non-enumerated set identifier from seen machines
VAR_G(expr)	Stack	Checks if given parameter is a concrete variable introduced by the operation refinement process. Added when the concrete variable is introduced
VAR_LOC(expr)	Stack	Checks if given parameter is a local variable introduced by current operation refinement. Added when the local variable is introduced

Figure 13 : Bart expression guards

IV.1.2 Predicate guards

Guard	Type	Description
PR(pred)	Calculated	Checks if given predicate is true using AtelierB prover pred must be a simple predicate with no guards
bisfalse(pred)	Calculated	Checks if not(pred) is present within the hypothesis stack. pred must be a simple predicate with no guards
bistrue(pred)	Calculated	Checks if pred constraint can be matched against

		hypothesis
		pred must be a simple predicate with no guards
bnot(pred)	Calculated	Checks if given constraint can not be checked against hypothesis
		pred can contain guards

Figure 14 : Bart predicate guards

IV.1.3 Substitution guards

Some of following guards are called substitution guards because their parameter is internally represented as substitution by the tool.

Guard	Type	Description
DECL_OPERATION(oper)	Stack	"oper" must be an operation description that can contain jokers. The shape of the parameter is prototype separated of operation body with a " ", for example: @h <-- @i(@j) PRE @p THEN @h := @g(@j) END. A guard of this type is added for each operation of machines seen by the component to refine.
bhasflow(sub)	Calculated	Checks if given substitution contains flow (i.e. branch structures as IF or SELECT substitutions)
bsearch(pattern list result)	Calculated	Checks if <i>pattern</i> substitution is present in <i>list</i> substitution. If so, <i>result</i>, which must be an uninstantiated joker, takes the value of list without pattern occurrences.

Figure 15 : Bart substitution guards

IV.2 Predicate synonyms

In addition to Bart guard extensibility, which requires code writing and recompilation, Bart provides a mechanism to the user allowing to custom predicates that can be used in rule constraints.

This is done using a special theory, which must be put in rule files (cf. VI).

Syntax of the predicate theory is:

```
PredicateTheory
=
    "THEORY_PREDICATES"
    "IS"
    PredicateDefinition { "|" PredicateDefinition }
```

```

"END"
.
PredicateDefinition
=
    ident "(" JokerList ")" " " <=> " Predicate
.

```

Syntax 1 : Predicate theory

Here is an example of this special theory:

```

THEORY_PREDICATES IS
    test(@a) <=> bnum(@a) |
    NumOrIdent(@a) <=> bident(@a) or test(@a) |
    belongs(@a,@b) <=> (@a : @b) |
    ElementOfSet(@d) <=> @d : @s
END

```

Figure 16 : Predicate theory example

Left part of each line is a synonym. It is a predicate identifier with a list of jokers between parentheses. Right part is the value, it's a predicate containing jokers. When these keywords are found in a rule file, they are replaced by predicates described on the right part. Jokers present in the value and in the synonym are replaced by the element given at use. Others jokers are left unchanged.

For example, if the preceding predicate theory is used and a rule has the following constraint:

```
belongs(@c,INT) & 0 <= @c & ElementOfSet(@e)
```

, the following predicate will be actually loaded by Bart:

```
@c: INT & 0 <= @c & @e: @s
```

Every synonym predicate defined in the rule file must have been defined before. If, for example, Bart finds `test(@a)` before the predicate theory that defines `test`, it will load this predicate as a type predicate (predicates to be matched with hypothesis added by variable refinement, cf. VII.3).

A synonym can use another one previously defined (as `NumOrIdent` uses `test` in the example).

A predicate theory is local to its definition rule file. Definitions from a particular file can not be used in another one.

V PRAGMAS AND COMMENTS

In most cases, Bart tries to keep comments from original B component elements, and to rewrite them beside suitable refinement results.

Pragmas are special comments that the user writes in the B component to refine in order to impact the refinement process. These elements are not processed by AtelierB, but only by Bart. AtelierB processes them as simple comments. Each pragma begins with `/* pragma_b`.

V.1 EMPILE_PRE, DEPILE_PRE

These two pragmas are used to modify the top of Bart predicate stack. They must be written before a substitution of an operation from the machine to refine. They are used when the refinement of the substitution they are written before begins.

`/* pragma_b EMPILE_PRE(predicate) */` is used to add predicate at the top of the hypothesis stack.

`/* pragma_b DEPILE_PRE */` is used to remove last predicates added to the stack.

For example, if Bart must refine following substitution:

```

IF valeur > 100 THEN
  /* pragma_b EMPILE_PRE(valeur > 0) */
  Substitution1
ELSE
  /* pragma_b DEPILE_PRE */
  Substitution2
END
  
```

Figure 17 : Substitution for EMPILE_PRE and DEPILE_PRE example

Let's assume that Bart adds the if condition for refining the THEN branch, and the negation of the condition for refining the ELSE branch. Following table presents the stack state depending on pragmas presence.

Branch	Stack	
	Without pragma	With pragma
Then branch	valeur > 100 & Previous predicates	valeur > 0 & valeur > 100 & Previous predicates
Else branch	Not(valeur > 100) &	Previous predicates

	Previous predicates	
--	---------------------	--

Figure 18 : Example of stack evolution with *EMPILE_PRE* and *DEPILE_PRE*

V.2 Magic

Pragma MAGIC can be used to directly specify in B components which rules must be used to refine certain elements (variables or substitutions). It is useful to force use of a certain rule. The given one is used even if suitable rules could be found before it in a regular rule research. Bart checks that the given rule can be applied to the element (pattern matching and constraint checking).

V.2.1 For variables

Magic pragma is used to specify which rule should be used to refine a certain variable. The syntax is `/* pragma_b MAGIC(theory.rule,variable) */`. It means that given rule from given theory will be used to refine the variable.

Variable magic pragmas must be put at the machine beginning. There can be several magic pragmas at the machine beginning. As the rule file is not specified, Bart processes rule files in the classic rule research order to find the rule in the suitable theory. If no such rule is found, a refinement error occurs.

V.2.2 For substitutions

Magic pragma can also be used for refining substitutions. The pragma must be written directly before the involved substitution in the B model. The syntax is `/* pragma_b MAGIC(theory.rule) */`

For example:

```
/* pragma_b MAGIC(theory_operation.r_affect_bool) */
bool_value := TRUE
```

will refine the substitution using `r_affect_bool` rule in theory `theory_operation`.

V.3 CAND

This pragma has a particular shape. It must be written `/* CAND */`, and be put just before a “&” operator in B model.

It means that this operator is a conditional and (right part is not evaluated if left part is false).

A “&” operator from B model that has a `/* CAND */` pragma will match with `cand` operator of Bart rule files.

VI RULE FILES

VI.1 Syntax

Rule files are files containing theories, each theory containing one or several rules used to refine given component. Rule file extension is usually .rmf.

A rule file can contain variable, operation, structure and initialisation theories. It can also contain utility theories such as tactic, user pass, or definition of predicates synonyms.

Syntax of rule files is:

```
RuleFile = [ Theory { "&" Theory } ].  
  
Theory  
=  
VariableTheory  
| OperationTheory  
| StructureTheory  
| InitialisationTheory  
| UserPassTheory  
| TacticTheory  
| PredicateTheory  
.
```

Syntax 2 : Rule files

The rule file syntax must also respect certain constraints:

- User pass can be present at most once
- Tactic can be present at most once
- Predicate theory can be present at most once

Order between theories has no syntactical impact, except for predicates theory: it must be defined before its elements are used in the rule.

Order between theories has an impact on the rule research, as the standard process (no user pass or tactic) reads theories from bottom to top.

User pass and tactic can be defined anywhere in the file, even before theories they refer to have been defined.

VI.2 Using rule files

VI.2.1 Providing rule files on command line

As it was previously described, user provides rules files when launching Bart by using `-r` parameter. This parameter can be present several times, and is not mandatory.

When it searches for rules, Bart processes rule files from right to left, according to command line order.

Let's consider following command line:

```
./bart -m machine.mch -r rule2.rmf -r rule1.rmf
```

For a given element to refine, the tool will search first in `rule1.rmf`, and then in `rule2.rmf` if the first file did not contain a suitable rule.

VI.2.2 Rule file associated to the component

If directory that contains the given machine file also contains a rule file with same name, it has not to be specified on the command line, Bart will automatically load it.

If such a file is present, it will be used in priority (as it had been given last using `-r` parameter on command line).

For this command line:

```
./bart -m machine.mch -r rule.rmf
```

, Bart will look for `machine.rmf` in current directory. If it is present, rule files will be used in this order: `machine.rmf`, then `rule.rmf`.

VI.2.3 Bart refinement rule base

The tool comes with a set of predefined rule base, contained in the file `PatchRaffiner.rmf` present in Bart distribution. It provides rules that permit to refine most of the classical B substitutions.

When Bart is used on command line, the rule base must be provided using `-r` parameter.

The classical automatic refinement scheme is the following: most elements of given component can be refined using the rule base. If an element can not be refined with it, or needs a more specific treatment, user should write suitable rules in `rmf` files that will be provided after the rule base on command line, or in the component associated rule file.

VII VARIABLES REFINEMENT

VII.1 Variable theories syntax

```

VariableTheory
=
    "THEORY_VARIABLE" ident
    "IS"
    VariableRule { ";" VariableRule }
    "END" ident
.

VariableRule
=
    "RULE" ident ["(" JokerList ")"]
    "VARIABLE" JokerList
    [ "TYPE" ident ["(" JokerList ")"]
    [ "WHEN" Predicate ]
    "IMPORT_TYPE" Predicate
    (VariableImplementation | VariablesRefinement )
    "END"
.

VariableImplementation =
    "CONCRETE_VARIABLES" JokerList
    [ "DECLARATION" Predicate ]
    "INVARIANT" Predicate
.

VariablesRefinement =
    "REFINEMENT_VARIABLES"
    VariableRefinement { "," VariableRefinement }
    "GLUING_INVARIANT" Predicate
.

VariableRefinement =
    "CONCRETE_VARIABLE" joker
    "WITH_INV" Predicate
    "END"
    |
    "ABSTRACT_VARIABLE" joker
    "REFINED_BY" ident "." ident "(" Expression ")"
    "WITH_INV" Predicate
    "END"
.

```

Syntax 3 : Variable rule theories

Each theory has an identifier, which must be repeated after the END keyword. A theory can contain several rules, each rule having its own unique identifier. Each following subsection will associate a variable refinement functionality with one or more clauses of variable rules.

VII.2 Variable rule research

Variable rule research is different from rule research for operations and initialisation. Instead of processing each variable and finding a suitable rule for it, it processes each rule of considered theories (all variable theories or a subset if tactic or user pass is used, cf. IX) and checks if it can be used to refine some variables.

This is necessary because a single variable rule can be used to refine several variables. Once a rule has been selected for one (or several) variable, resulting refinement variables can be calculated from its clauses.

The principle of rule research is the following:

- At the beginning the tool considers the set of abstract variables to refine
- It processes every theory that could be used (according to tactic, user pass or neither) from bottom to top. For each theory:
 - The tool processes all rules of theory from bottom to top. For each variable rule:
 - Bart determines which variables can be refined by current rule
 - Refined variables are removed from the set of remaining variables

Figure 19 : Processing variable theories to find rules

This process stops when there are not variables to refine anymore, or when all variable rules to consider have been treated. Variable refinement is successful if all variables have been associated with a rule. It is a failure if all rules have been treated and some variables could not be refined.

For a certain rule, Bart determines which variables it can refine as follow:

- The tool tries every combination of values to instantiate joker list of VARIABLE clause. For each instantiation:
 - Bart checks constraint expressed in WHEN clause against hypothesis stack, with jokers of VARIABLE clause instantiated
 - If WHEN constraint could be checked, variables used to instantiate VARIABLE clause can be refined by this rule
 - Variable refined by the rule are removed from set of remaining variables, to be sure they won't be used in following tried instantiation

Figure 20 : Searching variables refined by a particular rule

If current rule has several jokers in VARIABLE clause, there are more combinations to try than for simple variable rules.

Following example presents results of a variable rule research, with given theories and predicates stacks. Variable to refine are {aa, bb, cc, dd, ee}.

Theories	Stack
THEORY_VARIABLE t1 IS	aa : INT &

<pre> RULE r1 VARIABLE @a WHEN @a : INT & @b < @a & ABCON(@b) [...] END; RULE r2 VARIABLE @a, @b WHEN @a : INT & @b : NAT & @a < 0 & 0 <= @b [...] END END t1 & THEORY_VARIABLE t2 IS RULE r3 VARIABLE @a WHEN @a : NAT & 1 <= @a [...] END END t2 </pre>	<pre> bb : INT & cc : NAT & dd : NAT & ee : NAT & value < aa & bb < 0 & 0 <= cc & 1 <= dd & 1 <= ee & ABCON(value) </pre>
--	--

Figure 21 : Theories and stack for variable rule research example

For this stack, the rule research process is the following:

- Trying theory t2
 - Trying rule r3
 - Possible instantiations of VARIABLE clause: {aa}, {bb}, {cc}, {dd}, {ee}
 - dd and ee can be refined by the rule
 - Set of variables to refine is now {aa, bb, cc }
- Trying theory t1
 - Trying rule r2
 - Possible instantiations of VARIABLE clause : {aa, bb}, {bb, cc}, {aa, cc}, {bb, aa}, {cc, bb}, {cc, aa}
 - Only {bb,cc} instantiation makes the WHEN clause be checked. bb and cc can be refined by the rule
 - Set of variables to refine is now {aa}
 - Trying rule r1
 - aa can be refined by current rule

Figure 22 : Variable rule research example

For this example, variable refinement is successful, as each variable has been refined.

VII.3 Storing information predicates about found variable rules

Bart allows specifying, in variable rules, predicates that will be added in the stack if the rule is selected. These predicates will be called “Type predicates” and are specified in TYPE clause of variable rules.

Type predicates are constituted of an identifier and a joker list between parentheses. They are added to the stack after variables refinement, to be reused in operations and initialisation refinement (in substitution rules constrains). Jokers of the joker list must have been present in VARIABLE or WHEN clauses, because they have to be instantiated for the type predicate to be added to the stack.

If we reuse previous example and complete rules with these TYPE clauses:

<pre> RULE r1 VARIABLE @a TYPE COMP(@a,@b) WHEN @a : INT & @b < @a & ABCON(@b) [...] END </pre>	<pre> RULE r2 VARIABLE @a, @b TYPE DOUBLE(@a,@b) WHEN @a : INT & @b : NAT & @a < 0 & 0 <= @b [...] END </pre>	<pre> RULE r3 VARIABLE @a TYPE SCALAR(@a) WHEN @a : NAT & 1 <= @a [...] END </pre>
--	---	---

Figure 23 : Rules for type predicate example

These predicates will be added after variable refinement previously described:

```

{SCALAR(ee) &
SCALAR(dd) &
DOUBLE(bb,cc) &
COMP(aa,value)}

```

Figure 24 : Example of type predicates adding

VII.4 Invariant for refined abstract variables

In the output chain components, refined abstract variables won't be necessarily implemented in the first one. It is necessary to provide the invariant that must be copied in component in which variables refined by the rule have not been implemented yet.

This is done within the clause IMPORT_TYPE of the rule. This clause is a predicate which may contain jokers. These jokers must have been present in VARIABLE or WHEN clauses, because they have to be instantiated for the predicate to be copied in output components.

VII.5 Specifying variable refinement results

Refinement results for abstract variables are specified in REFINEMENT_VARIABLES or CONCRETE_VARIABLES clauses of variables rules. These two clauses can not be used at the same time.

VII.5.1 Using CONCRETE_VARIABLES clause

Using CONCRETE_VARIABLE clause is simpler than using REFINEMENT_VARIABLES, but it is less powerful as it is impossible to specify abstract refinement variables. This clause corresponds to the VariableImplementation element of the syntax presented in VII.1.

This clause contains a list of jokerized identifiers (CONCRETE_VARIABLES clause), which will be concrete variables refining abstract variable treated by the rule, and the invariant that will be added for these concrete variables (INVARIANT clause). Jokers in the invariant must have been instantiated during the rule selection. Expressions designating new concrete variable must be built on previously instantiated jokers.

As the result is a joker list, and not a single joker, it is possible to specify several refinement variables for a unique rule.

For example, if the rule:

```

RULE r_ens
VARIABLE
  @a
TYPE
  raffinement_ensemble(@a, @b, @c)
WHEN
  SET(@c) &
  @a <: @c
IMPORT_TYPE
  @a <: @c
CONCRETE_VARIABLES
  @a_r
INVARIANT
  @a_r : @c --> BOOL &
  @a = @a_r~[{TRUE}]
END
  
```

Figure 25 : Example of variable refinement rule with variable implementation

is used to refine the abstract variable “ee”, this variable will be refined by “ee_r”. If we suppose that @c joker value determined by constraint checking was “set”, the following predicate will be added to the invariant of the output implementation it will be implemented in: ee_r : set --> BOOL & ee = ee_r~[{TRUE}].

VII.5.2 Using REFINEMENT_VARIABLES clause

Using this clause allows specifying both concrete and abstract variable to refine the abstract variable treated by the rule. It corresponds to VariablesRefinement element of syntax described in VII.1.

This clause must contain a list of VariableRefinement elements as described in the syntax in VII.1. Each one of these elements specify a refinement variable (abstract or concrete), and its associated invariant. For refinement abstract variables, rule that will be used to refine it must also be provided, with its parameters (cf. example). Jokers contained in subparts of these elements must all have been previously instantiated.

After the list of refinement variables, the GLUING_INVARIANT clause must be written. This predicate is the invariant that will be put in output components when all refinement variables will have been implemented. This predicate must only contain previously instantiated jokers.

Following rule using REFINEMENT_VARIABLES clause is equivalent to the previously described one:

```

RULE r_ens
VARIABLE
  @a
TYPE
  raffinement_ensemble(@a, @b, @c)
WHEN
  SET(@c) &
  @a <: @c
IMPORT_TYPE
  @a <: @c
REFINEMENT_VARIABLES
  CONCRETE_VARIABLE
    @a_r
  WITH_INV
    @a_r : @c --> BOOL &
  END
GLUING_INVARIANT
  @a = @a_r~[{TRUE}]
END

```

Figure 26 : Variable refinement rule with concrete variable

If we need to refine the variable with another abstract variable, the rule should be:

```

RULE r_ens
VARIABLE
  @a
TYPE
  raffinement_ensemble(@a, @b, @c)
WHEN
  SET(@c) &
  @a <: @c
IMPORT_TYPE
  @a <: @c

```

```

REFINEMENT_VARIABLES
  ABSTRACT_VARIABLE
    @a_r
  REFINED_BY
    theory.abstract_rule(@a_r)
  WITH_INV
    @a_r : @c --> BOOL &
  END
GLUING_INVARIANT
  @a = @a_r~[{TRUE}]
END

```

Figure 27 : Variable refinement rule with abstract variable

Here we directly specify which rule will be used to refine the new variable in REFINED_BY clause. The syntax is `theory.rule(parameters)`. Values specified between parentheses after the rule name are parameters. This means that the given rule must have parameters, like this:

```

RULE abstract_rule(@a)
VARIABLE @a
[...]
END

```

When a rule is given for a new refinement variable, the VARIABLE and WHEN clause jokers are instantiated with the variable name and the parameters. Then the regular rule checking process goes on as the WHEN constraint is verified.

If new abstract variables are introduced, a REFINEMENT component will be introduced in output chain.

VIII SUBSTITUTION REFINEMENT

Substitution refinement gathers operation, initialisation and structural rules. Operation and structure rules are identical. Initialisation rules are simpler versions of operation rules.

Syntax and principles of substitution refinement will be presented through operations rules. A further section will be dedicated to the different kind of substitution rules, their usage and differences. So in first sections of this chapter, “refining a substitution” will stand for “refining a substitution from an operation”.

Substitution refinement is more complex than variable refinement, as it can be recursive, i.e. result of refinement for a given substitution may have to be refined too. Furthermore, for a given substitution, refinement may need several sub-processes (cf. SUB_REFINEMENT clause or default refinement behaviours for parallel or semicolon). So refinement sub-branches are created and the underlying structure that can be used to represent substitution refinement is in fact a tree.

VIII.1 Rule syntax

Here is the syntax of operation theories:

```

OperationTheory
=
  "THEORY_OPERATION" ident
  "IS"
    OperationRule { ";" OperationRule }
  "END" ident
.

OperationRule
=
  "RULE" ident
  "REFINES" Substitution
  [ "WHEN" Predicate ]
  [ "SUB_REFINEMENT" SubRefinementRule { "," SubRefinementRule } ]
  ( "REFINEMENT" | "IMPLEMENTATION" )
    { RefinementVarDecl }
    Substitution
  [ "IMPLEMENT" IdentOrJokerList ]
  "END"
.

RefinementVarDecl =
  ("VARIABLE" | "ABSTRACT_VARIABLE" | "CONCRETE_VARIABLE") joker
  [ "REFINED_BY" ident "(" Expression ")" ]

```

```

"WITH_INV" Predicate
"WITH_INIT" Substitution
"IN"
.
SubRefinementRule =
  "(" Substitution ")" "->" "(" Joker ")"
.

```

Syntax 4 : Operation rule theories

Syntax for others kinds of substitution rules will be presented further.

As for variable theories, an identifier must be present after THEORY_OPERATION keyword and repeated after the END keyword.

VIII.2 Rule research

The substitution rule research process is simpler than for variables.

- For a substitution to refine, Bart processes each rule file as long as he could not find a rule.
- For each rule file it processes operation theories to consider (all theories, or a subset if tactic or user pass is used, cf. IX) from bottom to top.
- For each theory it processes operation rules from bottom to top
- For each rule, Bart checks if it can be used to refine currently treated substitution.

If each rule file was processed by Bart and no rule could be found for a certain substitution, an operation refinement may occur (cf. VIII.3).

Each operation rule has a pattern (REFINES clause) and may have a constraint (WHEN clause). The process used to check if a rule can be applied to a substitution is as described in III.3.4.

First the tool tries to match the rule pattern with the substitution. If it is successful, the rule can be applied under the condition it has no WHEN constraint or its WHEN constraint can be checked against hypothesis stack.

For example, if $\text{par_out} := \text{par_in1} + \text{par_in2}$ must be refined, with following theories and stack:

Theories	Stack
THEORY_OPERATION assign_plus IS RULE r_assign_plus_par_in REFINES @a := @b + @c WHEN PAR_OUT(@a) & PAR_IN(@b) & PAR_IN(@c) [...]	PAR_OUT(par_out) & PAR_IN(par_in1) & PAR_IN(par_in2)

<pre> END; RULE r_assign_plus_const REFINES @a := @b + @c WHEN PAR_OUT(@a) & ABCON(@b) & ABCON(@c) [...] END END assign_plus & THEORY_VARIABLE assign_minus IS RULE assign_minus_1 REFINES @a := @b - @c [...] END END assign_minus </pre>	
--	--

Figure 28 : Theories and stack for operation rule research

Let's suppose that for this rule file there is no tactic or user pass. The rule research will be as follow:

- First tried rule is assign_minus.assign_minus_1. Its pattern doesn't match the substitution, so it can not be used
- Second tried rule is assign_plus.assign_plus_const. Its pattern matches the substitution, but its WHEN constraint can not be checked, so it can not be used
- Third tried rule is assign_plus.assign_plus_par_in. Its pattern matches the substitution, and its WHEN constraint can be checked, so this rule is selected.

Figure 29 : Example of operation rule research

VIII.3 Refinement process

The substitution refinement process depends, for given rule and substitution, on the presence and content of SUB_REFINEMENT, IMPLEMENTATION and REFINEMENT clauses.

SUB_REFINEMENT clause corresponds to SubRefinementRule element of syntax described in VIII.1. It contains a "," separated list of sub-elements.

Each sub-element left part is a substitution that may contain jokers. These jokers must all have been instantiated by pattern matching and constraint checking. Right part of the sub-element must be a single and still uninstantiated joker.

This clause is used to refine the given substitution and store the result in given joker. This is done before calculation of the rule substitution result, so the sub-refinement can be used to express the result.

IMPLEMENTATION clause expresses the result of current rule. It contains a substitution which may contain jokers. All these jokers must have been

instantiated during pattern matching, constraint checking or sub-refinement processing. IMPLEMENTATION clause may also contain concrete operation refinement variable declaration (cf. VIII.6).

Using IMPLEMENTATION clause means that given result is the final result of current branch and doesn't need to be refined again.

REFINEMENT clause expresses the result of current rule. It contains a substitution which may contain jokers. All these jokers must have been instantiated during pattern matching, constraint checking or sub-refinement processing. REFINEMENT clause may also contain abstract or concrete operation refinement variable declaration (cf. VIII.6).

Using REFINEMENT clause means that given result is not the final result of current branch. The result of rule must be refined.

IMPLEMENTATION and REFINEMENT clause can not be both used in a same rule. When a rule has been selected (and eventual sub-refinements have been processed), the rule result is calculated by instantiating jokers of its result clause.

A rule can contain both SUB_REFINEMENT and REFINEMENT clauses. In this case, each subrefinement is calculated and stored in its joker. Then content of REFINEMENT clause is instantiated and refined.

For a substitution to refine, if no rule could be found, Bart will check if it can be refined using a "predefined behaviour". For some kinds of substitutions, Bart may know how to refine them if no rule is present. Predefined behaviour can be the end of current branch (skip substitution refinement) or a simple node of refinement tree. In this case, Bart may create one (BEGIN substitution refinement) or several (semicolon refinement) subnodes in refinement tree for current substitution. For each new subnode created by predefined refinement behaviour, the recursive refinement process is restarted as a rule or predefined behaviour will be searched for each one.

Following figure summarizes the process:

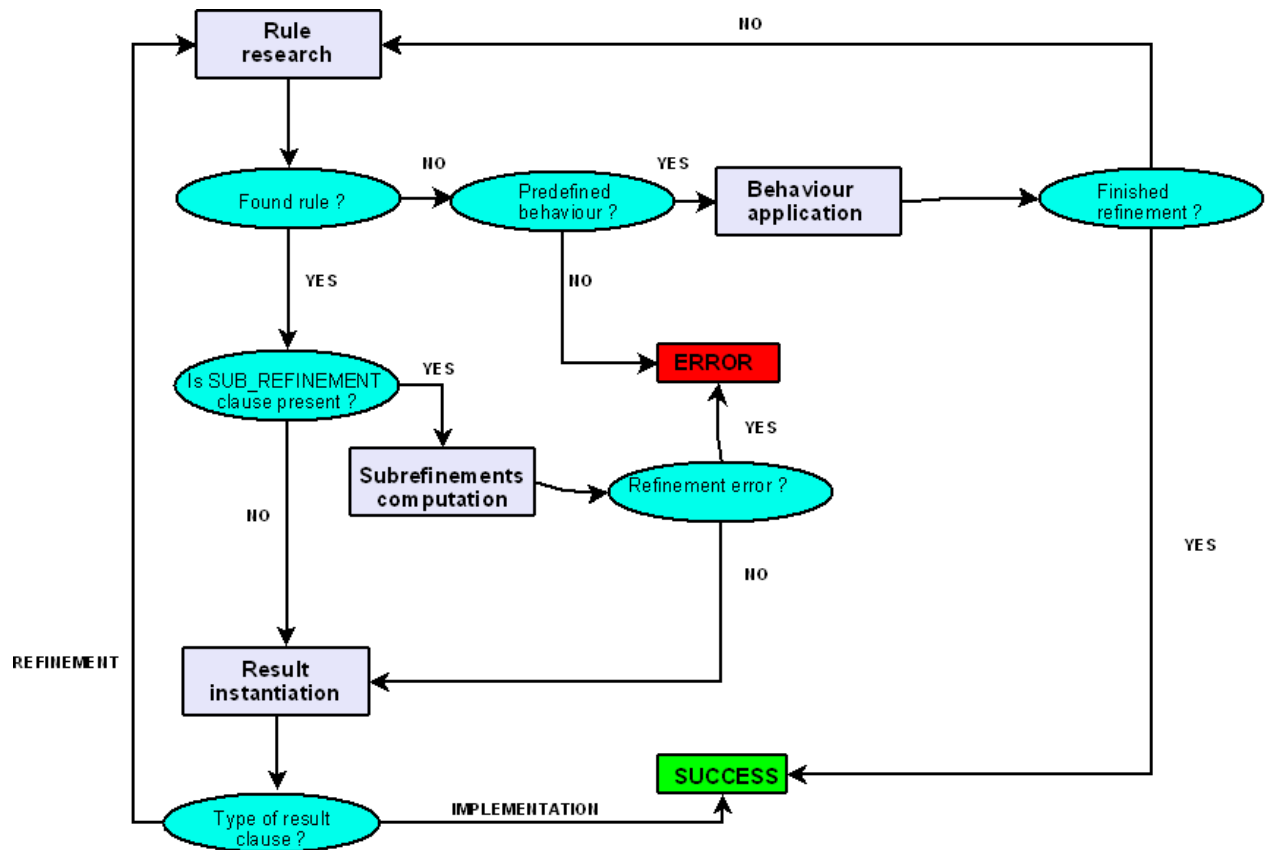


Figure 30 : Process of substitution refinement

Rectangles are actions processed by Bart. Ellipses are decisions. Error and success boxes represent error and success for current branch (an error in current branch means error in the whole refinement process).

Subrefinement computations are represented aside because they must be calculated for the result to be instantiated, but refinement of substitutions contained in left part of SUB_REFINEMENT clauses sub-elements uses the same process.

For example, if we consider following substitution to refine:

```

IF in < 0 THEN
  aa := aa + 1
ELSE
  aa := 0
END
  
```

and the following theories:

```

THEORY_OPERATION theory IS
  RULE assign
  REFINES
    @a := @b
  IMPLEMENTATION
  
```

```

    @a := @b
END;

RULE r_assign_plus_2
REFINES
    @a := @b + @c
IMPLEMENTATION
    @a := @b + @c
END;

RULE r_assign_plus
REFINES
    @a := @b + @c
WHEN
    bnot(B0EXPR(@a))
REFINEMENT
    #1 := @b + @c ;
    @a := #1
END ;

RULE r_if
REFINES
    IF @a THEN @b ELSE @c END
SUB_REFINEMENT
    (@b) -> (@d),
    (@c) -> (@e)
IMPLEMENTATION
    #1 := bool(@a);
    IF #1 = TRUE THEN @d ELSE @e END
END
END theory

```

Figure 31 : Substitution and theories for rule tree example

#x expressions written in rules result clauses are used to introduce local variables (cf. VIII.5.5).

For this example the resulting rule tree will be:

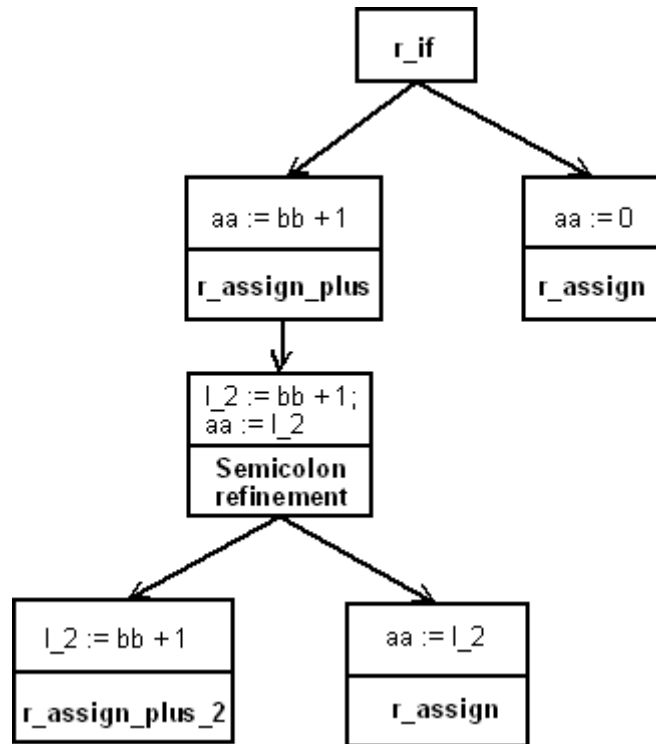


Figure 32 : Example of refinement rule tree

Each rectangle (except the first one which shows only the first found rule) shows the substitution to refine at current node, and the found rule.

First rule (r_if) has its result described in an IMPLEMENTATION clause but the refinement goes on as it contains SUB_REFINEMENT clauses. The refinement of r_assign_plus_rule_2 rule result uses the predefined refinement behaviour for semicolon.

For the refinement of this substitution, the result will be:

```

l_1 := bool(in < 0);
IF l_1 = TRUE THEN
    l_2 := bb + 1;
    aa := l_2
ELSE
    aa := 0
END
  
```

VIII.4 Default refinement behaviours

When a substitution must be refined and no rule could be found for it, Bart may apply a predefined behaviour to process refinement further.

If both a rule and a predefined behaviour are suitable for a substitution, the rule will be applied. For example Bart knows by default how to refine a semicolon substitution. But if a rule is present with @a;@b pattern and a WHEN constraint that can be checked for current substitution, Bart will use the rule.

Following table show which kind of substitutions can be refined by Bart even if no rule could be found in rule files. Here are only shown regular B substitutions that can be refined by predefined behaviours. Some Bart specific substitutions use this mechanism to control the refinement process, they will be described later.

In this table result(sub) means refinement result of substitution sub.

Substitution	Refinement result	Comment
Semicolon : sub1 ; sub 2	result(sub1); result(sub2)	-
Parallel: sub1 sub2	sequentialization(result(sub1) ; result(sub2))	Result is sequentialized. Variables modified in left part and read in right part are stored in local variables
Bloc substitution: BEGIN sub END	BEGIN result(sub) END	-
Guarded substitution: PRE predicate THEN sub END	BEGIN result(sub) END	"predicate" is added to the hypothesis stack for refining "sub"
Assertion substitution: ASSERT predicate THEN sub END	ASSERT predicate THEN result(sub) END	"predicate" is added to the hypothesis stack for refining "sub"
Operation call	Refined by itself	-
Skip	Refined by itself	-
Local variables : VAR list IN sub END	VAR list IN result(sub) END	VAR_LOC hypothesis is added to the stack for each element of "list"
Loop substitution: WHILE condition DO body INVARIANT I VARIANT V END	WHILE condition DO result(sub) INVARIANT I VARIANT V END	"condition" is added to the hypothesis stack for refining "body"

Figure 33 : Bart predefined refinement behaviours

As described in the table, a sequentialization is done when a parallel substitution is refined. For example if $aa := bb \parallel cc := aa$ must be refined and each branch is refined by itself, result without sequentialization would be $aa := bb ; cc := aa$, which is incorrect. So Bart makes sequentialization, and the real produced result will be $l_1 := aa ; aa := bb ; cc := l_1$, where l_1 is a local variable declared for the sequentialization. The local variable will be declared with others ones coming from # declaration in rules (cf. VIII.5.5).

VIII.5 Special refinement substitutions

In operation rules result clauses, it is possible to use Bart specific substitutions to control the refinement process or add elements to the produced result.

These substitutions don't exist in regular B models, and they can only be written in REFINEMENT or IMPLEMENTATION clauses of substitution rules and used to express the rule result. As they are present only in result clauses, all jokers contained in these substitutions must have been instantiated before. They are presented in following sections.

VIII.5.1 Iterators

Several substitutions can be used in Bart to manage iterators. These substitutions become WHILE loops when the result of rule they are written in is calculated. At the same time, some of them generate iterator machines that contains operations called in generated while loops. These generated machines are then refined by Bart using predefined rules.

VIII.5.1.1 TYPE_ITERATION

TYPE_ITERATION substitution allows specifying loops iterating on all elements of a set. In the produced implementation, this substitution is replaced by an automatically built WHILE loop which calls operations from an iteration machine created by Bart.

TYPE_ITERATION substitution syntax is as follow:

```
"TYPE_ITERATION" "("
  [ ("tant_que"|"while") ">=" IdentOrJokerOrVarDecl "," ]
  "index" ">=" Expression ","
  "type" ">=" Expression ","
  "body" ">=" "(" Substitution ")" ","
  "invariant" ">=" Predicate
  ")"
```

Syntax 5 : Type iteration

The different clauses meaning is:

- *while*: It must be given a variable (instantiated or not). This clause may be used if the iteration might be stopped before all elements of the set have been processed. If while clause is present, the loop continues as long as there are still more elements in the set, and given variable is TRUE. Given variable should be set to FALSE in the user defined loop body part to stop the loop
- *index* : Name of the variable that will contain each element of the given set
- *type* : Set the loop is iterating on
- *body* : User defined part of the loop body
- *invariant* : User defined part of the loop invariant

This shows how Bart generates the WHILE loop for a TYPE_ITERATION substitution (with no while clause):

```

vg_loop <-- init_iteration_TYPE ;
WHILE vg_loop = TRUE DO
    vg_loop, index <-- continue_iteration_TYPE ;
    body
INVARIANT
    vg_loop= bool(TYPE_remaining /= {}) &
    TYPE_remaining \/ TYPE_done = TYPE &
    TYPE_remaining /\ TYPE_done = {} &
    invariant
VARIANT
    card(TYPE_remaining)
END

```

Figure 34 : Type iteration generated loop, without while parameter

In this example, vg_loop is the automatically generated variable used to iterate on elements of the set. If a while clause is added to the TYPE_ITERATION substitution, generated loop becomes:

```

vg_loop <-- init_iteration_TYPE ;
WHILE vg_loop = TRUE DO
    vg_loop, index <-- continue_iteration_TYPE ;
    body ;
    vg_loop := bool(vg_loop = TRUE & while = TRUE)
INVARIANT
    vg_loop= bool(TYPE_remaining /= {}) &
    TYPE_remaining \/ TYPE_done = TYPE &
    TYPE_remaining /\ TYPE_done = {} &
    invariant
VARIANT
    card(TYPE_remaining)

```

END

Figure 35 : Type iteration generated loop, with while parameter

In these generated loops, called operations are defined in the following generated machine:

```

MACHINE
  iterator_name
ABSTRACT_VARIABLES
  TYPE_remaining, TYPE_done
INVARIANT
  TYPE_remaining <: TYPE &
  TYPE_done <: TYPE &
  TYPE_remaining /\ TYPE_done = {}
INITIALISATION
  TYPE_remaining := {} ||
  TYPE_done := {}
OPERATIONS

continue <-- init_iteration_TYPE =
BEGIN
  TYPE_done := {} ||
  TYPE_remaining := TYPE ||
  continue := bool( TYPE /= {} )
END;

continue, elt <-- continue_iteration_TYPE =
PRE
  TYPE_remaining /= {}
THEN
  ANY
    nn
  WHERE
    nn : TYPE &
    nn : TYPE_remaining
  THEN
    TYPE_done := TYPE_done \/ {nn} ||
    TYPE_remaining := TYPE_remaining - {nn} ||
    elt := nn ||
    continue := bool( TYPE_remaining /= {nn} )
  END
END

END

```

Figure 36 : Type iteration generated machine

This is a simple example in which a single iterator is generated for a given refined component. Generated machines can be more complex (cf. VIII.5.1.4)

VIII.5.1.2 INVARIANT_ITERATION

As TYPE_ITERATION, this substitution allows to automatically generate loops. But here, iteration is done on the image of a relation element.

INVARIANT_ITERATION syntax is:

```
"INVARIANT_ITERATION" "("
  [ ("tant_que"|"while") ">=" IdentOrJokerOrVardecl "," ]
  "1st" "index" ">=" Expression ","
  "2nd" "index" ">=" Expression ","
  "constant" ">=" Expression ","
  "1st" "type" ">=" Expression ","
  "2nd" "type" ">=" Expression ","
  "body" ">=" "(" Substitution ")" ","
  "invariant" ">=" "(" Predicate ")"
  ")"
```

Syntax 6 : Invariant iteration syntax

Clauses meaning is:

- *while* : If present, provides a variable which permits to interrupt the loop before its natural ending
- *constant* : Defines the relation which will be used to iterate
- *1st index*: Defines original element of iteration. Iteration will be done on constant[{1st index}]
- *2nd index* : Element storing current element of the loop
- *1st type* : Type of constant domain elements
- *2nd type* : Type of constant range elements
- *body* : User defined part of the loop body
- *invariant* : User defined part of the loop invariant

Generated loop for an INVARIANT substitution is:

```
vg_loop <- init_iteration_CONSTANT(index1 );
WHILE vg_loop = TRUE DO
  vg_loop, index2 <- continue_iteration_CONSTANT(index1 );
  body
INVARIANT
  vg_loop = bool(CONSTANT_remaining /= {}) &
  CONSTANT_remaining \ / CONSTANT_done = CONSTANT [{index1 }]
  CONSTANT_remaining /\ CONSTANT_done = {} &
  invariant
VARIANT
  card(CONSTANT_remaining)
END
```

Figure 37 : Invariant iteration generated loop

As for TYPE_ITERATION, `vg_loop := bool(vg_loop = TRUE & while = TRUE)` will be added to the loop body if a while substitution is added.

Generated iteration machine for a single INVARIANT_ITERATION is:

```

MACHINE
  iterator_name
ABSTRACT_VARIABLES
  CONSTANT_remaining,
  CONSTANT_done
INVARIANT
  CONSTANT_remaining <: ran(CONSTANT) &
  CONSTANT_remaining <: TYPE2 &
  CONSTANT_done <: TYPE2 &
  CONSTANT_remaining /\ CONSTANT_done = {}
INITIALISATION
  CONSTANT_remaining := {} ||
  CONSTANT_done := {}
OPERATIONS

continue <-- init_iteration_CONSTANT(elt) =
PRE
  elt : TYPE1
THEN
  CONSTANT_done := {} ||
  CONSTANT_remaining := CONSTANT[{elt}] ||
  continue := bool(CONSTANT[{elt}] /= {})
END;

continue, elt <-- continue_iteration_CONSTANT =
PRE
  CONSTANT_remaining /= {}
THEN
  ANY
    nn
  WHERE
    nn : TYPE2 &
    nn : CONSTANT_remaining
  THEN
    CONSTANT_done := CONSTANT_done \/ {nn} ||
    CONSTANT_remaining := CONSTANT_remaining - {nn} ||
    elt := nn ||
    continue := bool(CONSTANT_remaining /= {nn})
  END
END
END

```

Figure 38 : Invariant iteration generated machine

VIII.5.1.3 CONCRETE_ITERATION

CONCRETE_ITERATION substitution also produces automatically generated WHILE loops. Unlike TYPE_ITERATION or INVARIANT_ITERATION, these loops don't use any iteration machine.

Syntax for CONCRETE_ITERATION substitution is:

```
"CONCRETE_ITERATION" "("
  "init_while" ">=" "(" Substitution ")" ", "
  ("tant_que"|"while") ">=" Expression ", "
  "body" ">=" "(" Substitution ")" ", "
  "invariant" ">=" "(" Predicate ")" ", "
  "variant" ">=" Expression ", "
  "flag" ">=" IdentOrJoker
  ")"
```

Syntax 7 : Concrete iteration

The generated loop for this substitution is:

```
init_while ;
vg_loop := bool( while );
WHILE vg_loop = TRUE DO
  /*? Flag iteration: flag ?*/
  body ;
  vg_loop := bool( while )
INVARIANT
  invariant
VARIANT
  variant
END
```

Figure 39 : Concrete iteration generated loop

VIII.5.1.4 Iteration components

During refinement process, Bart stores information about iteration machines used by operations refinement and defined by TYPE_ITERATION or INVARIANT_ITERATION substitutions.

After splitting refinement results in output components (cf. X), Bart creates an iteration machine associated to each generated implementation, if necessary. Each iteration machine generated contains variables and operations for all iterators defined and used by refinement of operations implemented in associated implementation.

Following table presents which abstract variables and operations are generated in iteration machines for the refinement of a component "Machine", according to TYPE_ITERATION and INVARIANT_ITERATION substitutions used during refinement.

Iterators used by operations	Associated iteration machine
Machine_i	-
Operation1: No iterator defined	
Machine1_i	Machine1_{it}
Operation2: Type iterator on type 1 Invariant iterator on const1 Operation3: Type iterator on type2 Operation4: Invariant iterator on const1	Abstract variables: type1_remaining, type1_done, const1_remaining, const1_done, type2_remaining, type2_done Operations: init_iteration_type1 ; continue_iteration_type1; init_iteration_const1; continue_iteration_const1; init_iteration_type2; continue_iteration_type2
Machine2_i	Machine2_{it}
Operation5: Type iterator on type2 Operation6: Invariant iterator on const2	Abstract variables: type2_remaining, type2_done, const2_remaining, const2_done Operations: init_iteration_type2; continue_iteration_type2; init_iteration_const2; continue_iteration_const2

Figure 40 : Example of generated iterators

If a same iterator is used by several operations of implementation, it is only created once in iteration machine. Bart gathers all iteration variables and operations necessary for all refinement results written in the implementation. Invariant and initialisation are generated according to defined variables. Real iteration machines are actually merges of iteration machines presented in VIII.5.1.1 and VIII.5.1.2.

VIII.5.2 Using operations from seen machines - SEEN_OPERATION

SEEN_OPERATION substitution is used to insert a call to an operation from a seen machine in the rule result. Its syntax is:

```
"SEEN_OPERATION" "("
  "name" ">=" IdentOrJoker ","
  "out" ">=" "(" [ IdentJokerVardeclList ] ")" ","
  "in" ">=" "(" [ IdentJokerVardeclList ] ")" ","
  " body " ">=" "(" Substitution ")"
"
```

Syntax 8 : Seen operation

- name : Name of the operation to use
- out : Output parameters of the operation call

- in : Input parameters of the operation call
- body: Substitution that may be used by Bart to control in seen machines that it corresponds to the given identifier. For now the control is not done, so the clause can be filled with @_ joker

For example,
 SEEN_OPERATION(
 name => operation,
 out => (out1),
 in => (in1),
 body => (@_))

will be converted in out1 <-- operation(in1) operation call.

When SEEN_OPERATION is used, Bart doesn't check if the operation exists or if the user has provided the correct number of parameters. The operation is supposed to exist.

If the operation existence must be checked, it is better to use the DECL_OPERATION guard in the WHEN clause of the rule, and then express the result using jokers instantiated by constraint checking.

VIII.5.3 Defining imported operations - IMPORTED_OPERATION

IMPORTED_OPERATION substitution lets the user create a new operation that will be called in this one refinement and inserts a call to it. The newly created operation will be declared further in the output components chain. For example, if currently refined operation is implemented in Machine1_i, the new one will be first declared in Machine2, and implemented in Machine2_i or a further implementation.

In the generated implementation, IMPORTED_OPERATION will be replaced by a call to the created operation.

IMPORTED_OPERATION substitution syntax is:

```
"IMPORTED_OPERATION" "("
  [ "name" ">" ident "," ]
  "out" ">" "(" [ IdentJokerVardeclList ] ")" ","
  "in" ">" "(" [ IdentJokerVardeclList ] ")" ","
  "pre" ">" "(" Predicate ")" ","
  "body" ">" "(" Substitution ")"
  ")"
```

Syntax 9 : Imported operation

- *name*: This facultative clause can contain a base for generating the name of the new operation. If it is given, Bart may add number suffixes to the identifier to distinguish between different generated operation (as a rule can be selected several times)
- *output*: Output call parameters. Formal output parameters for the operation definition will also be generated from this list

- *input*: Input call parameters. Formal input parameters for the operation definition will also be generated from this list
- *pre* : User defined part of the precondition for the new operation
- *body* : The new operation body

VIII.5.3.1 Naming new operations

If name clause is given in IMPORTED_OPERATION, Bart will generate a unique name from it by adding a number suffix to the identifier.

Else Bart will use current operation name as a base, and will add number suffix to it. For example, IMPORTED_OPERATION substitution used in refinement of operation1 may generate operation1_1, operation1_2, etc.

As there can be several level of overlapped operations (ex: operation generates operation1 which generates others operations), Bart may add several numbers to an original pattern. To avoid conflicts in naming, it adds underscore after the first counter and before each counter greater than 10.

For example :

```
operation -> operation1 -> operation1_1 -> operation1_1_11
operation -> operation1 -> operation1_1 -> operation1_11 -> operation1_111
```

Figure 41 : Imported operation naming example

VIII.5.3.2 Operation parameters

The user can provide input or output parameters for the operation.

Following table presents an IMPORTED_OPERATION treatment in a simple case where instantiation is {@a = aa, @b = bb, @c= cc}. For this example we do not consider operation abstraction and hypothesis stack (cf. VIII.5.3.3).

Rule	Operation call	Generated operation
IMPORTED_OPERATION(name => add out => (@a), in => (@b,@c), pre => (@b : INT & @c : INT), body => (@a := @b + @c))	aa <-- add1(bb, cc)	out <-- add1(in1, in2) = PRE in1 : INT & in2 : INT THEN out <-- in1 + in2 END

Figure 42 : Simple imported operation example

In some case, "body" can contain, when instantiated, identifiers that are local variables or current operation parameters, and that are not directly put by user as parameters of new operation.

In these cases, generated operation would be incorrect, as these identifiers would be unknown in the machine the new operation will be declared in. So in these particular cases, Bart automatically adds inputs (for read ones) or output (modified ones) parameters to give these values to the newly defined operation.

Let's consider a new example with instantiation $\{@a = aa, @e = bb + cc\}$, in which bb and cc are input parameters of current refined operation, and without considering the hypothesis stack or operation abstraction:

Rule	Operation call	Generated operation
<pre>IMPORTED_OPERATION(name => add out => (@a), body => (@a := @e))</pre>	<pre>aa <-- add1(bb, cc)</pre>	<pre>out <-- add1(in1,in2) = BEGIN out <-- in1 + in2 END</pre>

Figure 43 : Imported operation example with parameters adding

As bb and cc are local parameters that can not be directly exported in new operation body, two input parameters are automatically added to new defined operation.

If bb and cc had been global variables, Bart would not have added input parameters, as they could have been directly exported.

Functionality of automatically adding parameters is used to avoid typecheck errors when a parameter is missing, when local parameters can not be identified because they are contained in a joker (as in the example, $@e = bb + cc$), or to help the user when the instantiated body clause is huge and contains a lot of identifiers.

However, it is still better when every parameter that should be present in "in" or "out" clauses is, so that user can have a better control of the refinement.

VIII.5.3.3 Imported operation preconditions

As it has been said before, user can provide to Bart a piece of invariant that will be added to the generated operation. But Bart also automatically adds predicates to the operation invariant.

These added predicates are:

- Preconditions of abstractions of currently refined operation
- Predicates added by the refinement process while refining current operation (LH substitution, guarded substitution)

These predicates correspond in fact to every predicates added to the stack since the beginning of current operation refinement. When they are added, they are filtered with identifiers appearing in the new operation body, so that only relevant predicates are added.

Bart, when adding those predicates, doesn't check if user has put some of them in its "pre" clause, so sometimes predicates can appear several times. Basic typing predicates are often automatically added as they are normally present in previous abstractions. "pre" clause of the substitution should better be used for more complex and specific predicates.

Here is an example of Bart automatic predicate adding. The instantiation is $\{@a = aa, @b = bb, @c = cc\}$. bb and cc are input parameters of current operation, and the stack contains $bb : \text{INTEGER} \ \& \ cc : \text{INTEGER}$ (coming for example from operation precondition).

Rule	Operation call	Generated operation
<pre> IMPORTED_OPERATION(name => add out => (@a), in => (@b,@c), body => (@a := @b + @a)) </pre>	<pre> aa <-- add1(bb, cc) </pre>	<pre> out <-- add1(in1, in2) = PRE in1 : INTEGER & in2 : INTEGER THEN out <-- in1 + in2 END </pre>

Figure 44 : Example of imported operation precondition adding

VIII.5.3.4 Imported operations refinement

Refinement of given component operations may introduce new imported operations.

Once all original operations have been refined, Bart processes new imported operations to refine them. If their refinement introduces new operations, the process goes on until there are no new operations.

VIII.5.4 Controlling the refinement process

Some substitution that user can write in result clauses are not really expressing the result but permits to control the following refinement.

VIII.5.4.1 IMPLEMENT

IMPLEMENT syntax is:

```
ImplementSubstitution = "IMPLEMENT" "(" Substitution ")".
```

Syntax 10 : Implement

When IMPLEMENT is present in a result clause it means that its content will be written in the result without being more refined.

Usage of IMPLEMENT only makes sense in a REFINEMENT clause, as refinement stops when result is expressed in an IMPLEMENTATION one.

For example if IMPLEMENT(aa := 1) is present in a rule clause, aa := 1 will be written without being more refined, while others parts of the result clause may have their refinement processed further.

VIII.5.4.2 LH

LH stands for "Local Hypothesis" substitution. Its syntax is:

"LH" Predicate "THEN" Substitution "END"

Syntax 11 : LH

It is not translated in B substitution by Bart when the result clause is instantiated, but it allows the user to add a hypothesis for refining given substitution.

As IMPLEMENT, LH usage doesn't make sense in IMPLEMENTATION clause. It can be used in REFINEMENT, and, unlike other substitutions presented in this section, in SUB_REFINEMENT clause.

For example, with the following elements:

Substitution to refine	Rule
<pre>IF val > 0 THEN aa := TRUE ELSE aa := FALSE END</pre>	<pre>RULE r_if REFINES IF @a THEN @b ELSE @c END SUB_REFINEMENT (LH @a THEN @b END) -> (@d), (LH not(@a) THEN @c END) -> (@e) IMPLEMENTATION #1 := bool(@a) ; IF #1 = TRUE THEN @d ELSE @e END END</pre>

Figure 45 : Substitution and rule for LH example

, following figure shows the evolution of the stack, if we suppose it is empty before applying the rule.

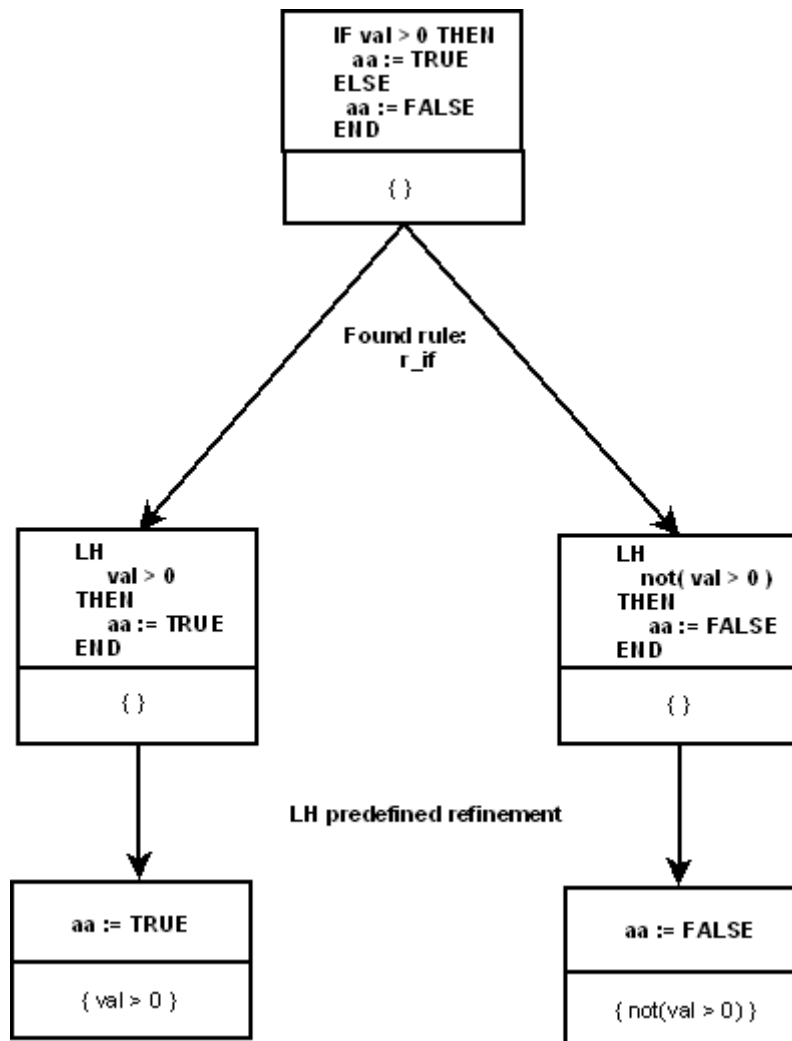


Figure 46 : Example of stack filling with LH substitutions

VIII.5.5 Local variable declarations

In Bart result clauses, it is possible to declare local variables and use them to express the result substitution.

The syntax is a “#” character followed by a number. If the same “#” declaration appears several times in the clause, it designates the same variable. Same declaration can be used in different rules, they will stand for different local variables.

If local variables are used by found rules during the whole operation refinement process, they will all be declared in a local variables (VAR...IN) substitution which will embrace the operation refinement result. For more information on formatting operation refinement results, see X.1.

Here is an example (we suppose that subrefinements are refinement by themselves):

Operation body	Rule	Operation refinement result
<pre> BEGIN IF value > 0 THEN Aa := TRUE ELSE Aa := FALSE END IF value2 > 0 THEN Bb := TRUE ELSE Bb := FALSE END END </pre>	<pre> RULE r_if REFINES IF @a THEN @b ELSE @c END SUB_REFINEMENT (LH @a THEN @b END) -> (@d), (LH not(@a) THEN @c END) -> (@e) IMPLEMENTATION #1 := bool(@a) ; IF #1 = TRUE THEN @d ELSE @e END END </pre>	<pre> VAR l_1, l_2 l_1 := bool(value > 0); IF l_1 = TRUE THEN Aa := TRUE ELSE Aa := FALSE END; l_2 := bool(value2 > 0); IF l_2 = TRUE THEN Bb := TRUE ELSE Bb := FALSE END END </pre>

Figure 47 : Local variable declaration example

VIII.6 Declaring operation refinement variables

Besides declaring local variables during operation refinement, it is also possible to declare new abstract or concrete variables that can be used in REFINEMENT clauses of rules.

This kind of declaration corresponds to RefinementVarDecl element of syntax described in VIII.1.

If VARIABLE or ABSTRACT_VARIABLE keyword is used, it means that the new variable is an abstract one. In this case, REFINED_BY clause may be present to specify which rule must be used. Syntax is REFINED_BY(theory.rule(parameters)). Parameters usage is identical as for abstract variable refinement. If REFINED_BY clause is not present, the rule for the variable will be simply searched in rule files.

If CONCRETE_VARIABLE is used, the new variable is a concrete one. In this case, usage of REFINED_BY clause doesn't make sense.

Invariant and initialisation for new variable are expressed in WITH_INV and WITH_INIT clauses.

If new abstract variables are introduced, a REFINEMENT component will be introduced in output chain.

VIII.7 Usage of substitution rules

VIII.7.1 Structural and operation rules - Operation refinement

VIII.7.1.1 Structural rules

Structural rules are exactly identical to operation rules, but they are gathered in theories called structure theories. So structure theories syntax is:

```
StructureTheory
=
  "THEORY_STRUCTURE" ident
  "IS"
    OperationRule { ";" OperationRule }
  "END" ident
.
```

Syntax 12 : Structure theories

Structural rules are only used in certain cases for refining operations of the given component to refine. Newly introduced imported operations are only refined with operation rules from operation theories.

VIII.7.1.2 Operation refinement process

Structural rules are used to refine operations from given component that contains control structures, i.e. at least one following substitutions: IF, SELECT. They are usually used to split IF and SELECT structure branches into several operation calls. Bart rule base contains structure theories allowing to treat these substitutions. But structure rules researching process is exactly identical to operation rules one, so user can define his own rules in his rule files.

Following figure shows how Bart uses structure and operation theories to refine operations of given component. This process is not used for refinement of created imported operations.

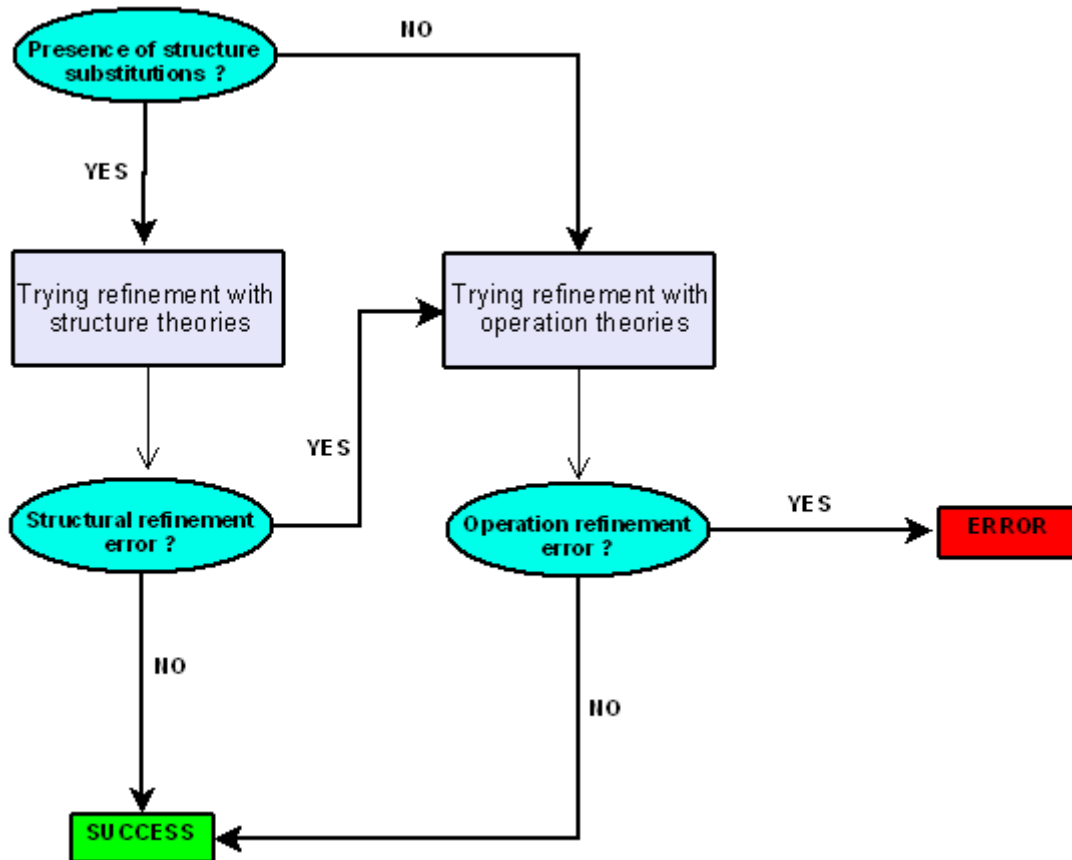


Figure 48 : Usage of structure and operation rules

Bart tries structural refinement if current operation contains structure substitution. A structural refinement error occurs if Bart can not find structure rules to completely refine the operation. If such an error occurs, Bart will try to refine the operation with operation rules from the beginning.

Consequently, an operation rule tree can contain only one kind of rules: structure rules (for operation containing structure that could be structurally refined) or operation rules (for operations without structure, or operations with structure that could not be structurally refined).

Once all original operations have been refined using structure or operation rules, imported operations introduced by this process are refined using exclusively operation rules.

Here is an example of refinement using structure and operation rules.

Rules	Operations
THEORY_STRUCTURE structure IS RULE default REFINES @a WHEN	operation(val) = PRE val : INTEGER THEN IF val > 0 THEN aa := TRUE

<pre> bnot(bhasflow(@a)) IMPLEMENTATION IMPORTED_OPERATION(out => (), in => (), pre => (0=0), body => (@a)) END; RULE if_then_else REFINES IF @a THEN @b ELSE @c END REFINEMENT #1 := bool(@a); IF #1 = TRUE THEN LH @a THEN @b END ELSE LH not(@a) THEN @c END END END END structure & THEORY_OPERATION operation IS RULE r_affect_bool_2 REFINES @a := @b WHEN match(@b,TRUE) or match(@b,FALSE) IMPLEMENTATION @a := @b END; RULE r_affect_bool_1 REFINES @a := @b WHEN (match(@b,TRUE) or match(@b,FALSE)) & bnot(BOEXPR(@a)) IMPLEMENTATION #1 := @b @a := #1 END; END operation </pre>	<pre> ELSE aa := FALSE END END; out <-- affect_true = BEGIN out := TRUE END </pre>
---	--

Figure 49 : Theories and operation for operation refinement example

For these rules and operations, refinement results are:

Operation	Found rules	Produced result
operation	- Guarded substitution refinement - structure.if_then_else - LH refinement - structure.default - LH refinement - structure.default	<pre> operation(val) = VAR l_1 IN l_1 := bool(val > 0); IF l_1 = TRUE THEN operation1 ELSE operation2 END </pre>

		END
affect_true	- operation.r_affect_bool_2	out <-- affect_true = BEGIN out := TRUE END
operation1 = BEGIN aa := TRUE END	- operation.r_affect_bool_1 - operation.r_affect_bool_2	operation1 = VAR l_1 IN l_1 = TRUE; aa := l_1 END
operation2 = BEGIN aa := FALSE END	- operation.r_affect_bool_1 - operation.r_affect_bool_2	operation2 = VAR l_1 IN l_1 = FALSE; aa := l_1 END

Figure 50 : Operation refinement example

Here we supposed hypothesis stack did not contain any predicates concerning aa variable, so that generated imported operation don't have preconditions.

VIII.7.2 Initialisation rules

For refining the initialisation of treated component, Bart uses special substitution rules called initialisation rules, gathered in initialisation theories. Initialisation rules are restricted substitution rules.

Initialisation rules syntax is presented hereafter:

```

InitialisationTheory
=
  "THEORY_INITIALISATION" ident
  "IS"
    InitialisationRule { ";" InitialisationRule }
  "END" ident
.

InitialisationRule
=
  "RULE" ident
  "REFINES" Substitution
  [ "WHEN" Predicate ]
  "IMPLEMENTATION" Substitution
  "END"
.

```

Syntax 13 : Initialisation theories

Bart refines the given component initialisation as it would refine an operation body, but with using initialisation rules instead of structure and operation rules.

Restrictions in initialisation rules in comparison to other substitution rules are:

- Initialisation rule result can only be specified in an IMPLEMENTATION clause. So an initialisation rule is always terminal
- Usage of subrefinements (SUB_REFINEMENT clause) is not allowed in initialisation rules
- Introduction of new global variables is not allowed in initialisation rules
- Introduction of local variables is not allowed in initialisation rules

When Bart refines initialisation, it usually goes down in the substitution by applying parallel predefined refinement behaviour, and searches for rules for each atomic initialisation element. So result of initialisation refinement is often a semicolon separated list of atomic substitutions.

When output components are generated, Bart splits initialisation in elementary elements. Each elementary element is an initialisation for a given variable. When a refinement variable is implemented in an output component, its associated initialisation element is also written.

As it is split and dispatched along output components, initialisation of given component to refine must be a parallel or semicolon separated list of elementary elements, each elementary element initialising a unique abstract variable.

Let's consider following initialisation and rules:

Initialisation	Rules
<pre>aa := 1 bb :: INTEGER cc :: BOOL</pre>	<pre>THEORY_INITIALISATION init IS RULE scalar_ini1 REFINES @a := @b WHEN SCALAR(@a) & B0(@b) IMPLEMENTATION @a := @b END; RULE scalar_ini2 REFINES @a :: @b WHEN SCALAR(@a) & PR(0 : @b) IMPLEMENTATION @a := 0 END; RULE scalar_ini3 REFINES @a :: @b WHEN SCALAR(@a) & PR(FALSE : @b) IMPLEMENTATION @a := FALSE END END init</pre>

Figure 51 : Substitution and theories for initialisation refinement example

where aa, bb, cc are abstract variables refined with a variable rule adding SCALAR predicate. Following table shows the results:

Found rule	Initialisation elementary element
Parallel refinement Parallel refinement Init.scalar_ini1 Init.scalar_ini2 Init.scalar_ini3	aa := 1 bb := 0 cc := FALSE

Figure 52 : Initialisation refinement example

IX TACTIC AND USER PASS THEORIES

Tactics and user passes should be used in Bart rule files to control the rule research process, and to avoid a processing of all rules from all theories. These theories are local to their rule files. Bart processes each rule file to find rules. For the rule file currently processed, it may use tactic or user pass to filter its theories to use.

IX.1 User pass theory

IX.1.1 Syntax

```
UserPassTheory
= "
USER_PASS" "
IS"
[ ("VARIABLE"|"OPERATION"|"INITIALISATION") ":" "(" IdentList ")" ]
{ ";" ("VARIABLE"|"OPERATION"|"INITIALISATION") ":" "(" IdentList ")" }
"END"
```

Syntax 14 : User pass theory

IX.1.2 Usage

User pass theory is used to specify, for different types of elements to be refined, which theories must be considered by Bart. There must be at most one user pass theory in a rule file.

Theories of a particular user pass element are considered from right to left.

For example, if following user pass is used:

```
USER_PASS IS
    VARIABLE : (tv1,tv2);
    OPERATION : (to1)
    INITIALISATION : (ti1, ti2)
END
```

Figure 53 : User pass theory example

, Bart will search variable rules only in theories tv1 and tv2, operation rules only in theory to1, and initialisation rules in theories ti1 and ti2.

At least one element (variable, initialisation or operation pass) must be present in user pass theory. At most one pass must be present for each kind of element.

If there are several user passes for a same kind of element, an error or a warning will be raised.

IX.2 Tactic theory

IX.2.1 Syntax

```

TacticTheory
=
    "THEORY" "TACTICS" "IS"
        { Tactic }
    "END"
.

Tactic =
    "VARIABLE" ":" VariableTactic { ";" VariableTactic }
|
    "INITIALISATION" ":" SubstitutionTactic { ";" SubstitutionTactic }
|
    "OPERATION" ":" SubstitutionTactic { ";" SubstitutionTactic }
.

SubstitutionTactic =
    IdentList ">=" "(" Substitution ")"
.

VariableTactic =
    IdentList ">=" "(" Predicate ")"
.
  
```

Syntax 15 : Tactic theory

IX.2.2 Usage

Tactics allow indicating which theories must be used for elements by using patterns. There must be at most one tactic theory in a rule file.

There are several sections for different elements to refine (variables, initialisation, and operations). At least one section must be present in the tactic, and each section should be present at most one time.

Each section contains a list of tactic elements, each one containing a theory list associated with a pattern. When an element must be refined by using the tactic theory, Bart processes the suitable tactic section from bottom to top, and tries to match the element with the pattern. If the variable or substitution to refine matches a tactic element pattern, rules for refining it are searched in the associated list of theories.

When a tactic pattern is selected, its theories are processed from right to left.

For example, if following tactic is used:

```

THEORY TACTICS IS
  VARIABLE :
    standard => (@a)
  INITIALISATION :
    iterateur_i, standard_i => (@a)
  OPERATION :
    assign_a_b, assign_a_b_2 => (@a := @b);
    assign_a_b_plus => (@a := @b + @c);
    assign_a_union_b_c => (@a := @b \ / @c);
END

```

Figure 54 : Tactic theory example

, when Bart must refine a variable, it will search for rules in theory standard. When it must refine initialisation, it will search for rules in iterateur_i and standard_i theories.

If $aa := set1 \setminus set2$ must be refined in an operation, assign_a_union_b_c theory will be used.

Note: If a pattern is selected and no rule is found (and no predefined behaviour), there will be a refinement error. Bart won't process the tactic further to check if the element to refine matches with other patterns. For example, with previous tactic, if $aa := bb + cc$ must be refined, and no rule is found in assign_a_b_plus, it won't search for rules in assign_a_b and assign_a_b_2 theories.

IX.3 Priority of Tactic and User pass theories

This section presents which theory will be used for a rule file according to the presence of tactic or user pass theories.

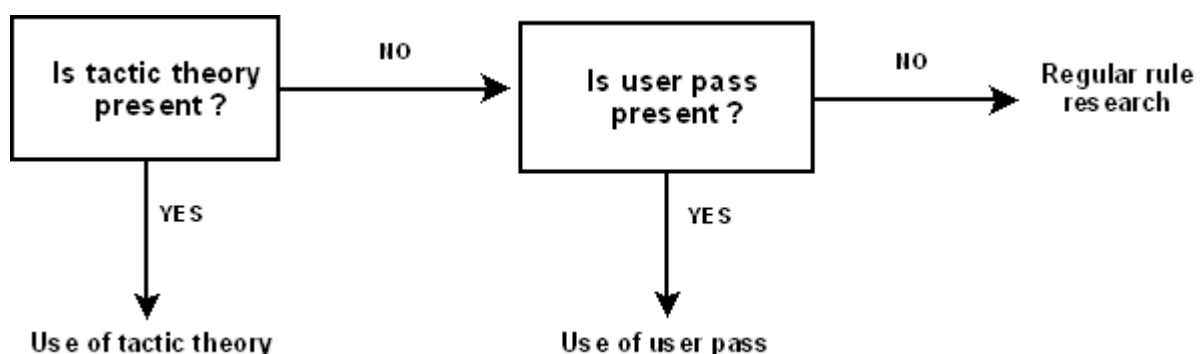


Figure 55 : Usage of tactics and user passes

This means that if tactic and user pass theory are both present, the tactic will be used.

When Bart has determined which kind of rule research (tactic, user pass or regular) will be used, it will only use this one, even if a refinement error occurs because no rule and predefined behaviour could be found. For example, if Bart uses user pass theory and a variable couldn't be refined, it won't try to find a rule in variable theories that were not included in the variable user pass.

X RESULT PRODUCTION AND WRITING

X.1 Formatting the result

To refine an operation, Bart launches its recursive rule research process on the operation substitution body.

At the end, the tool may apply a certain treatment on the produced result to write it as an operation body of output components. Furthermore, formatting process may also include introduction of a local variable substitution to declare local variables from this operation refinement (declared with the # syntax).

Following table shows how refinement results are formatted depending on the presence of new local variables. Generic elements are expressed with jokers here.

Refinement result	Declaration of local variables	Formatted result
<pre>PRE @p THEN @s END</pre>	No	<pre>PRE @p THEN @s END</pre>
	Yes	<pre>PRE @p THEN VAR @v IN @s END END</pre>
<pre>ASSERT @p THEN @s END</pre>	No	<pre>ASSERT @p THEN @s END</pre>
	Yes	<pre>ASSERT @p THEN VAR @v IN @s END END</pre>
<pre>BEGIN @b END</pre>	No	<pre>BEGIN @b END</pre>
	Yes	<pre>VAR @v IN @b END</pre>
<pre>VAR @l IN</pre>	No	<pre>VAR @l IN</pre>

@s END (local variables directly introduced by rules – not with # declaration)	Yes	@s END VAR @v IN VAR @l IN @s END END
@s (other substitutions)	No	BEGIN @s END
	Yes	VAR @v IN @s END

Figure 56 : Bart refinement result formatting

Here are examples of Bart result formatting:

Operation	Refinement result	Declared variables	local	Formatted result
affect_sum(in1,in2) = PRE in1 : INTEGER & in2 : INTEGER THEN abvar := in1 + in2 END	BEGIN l_1 := in1 + in2; abvar := l_1 END	l_1		affect_sum(in1,in2) = VAR l_1 IN l_1 := in1 + in2; abvar := l_1 END
out ← lire_abvar = out := abvar	out := abvar	-		BEGIN out := abvar END

Figure 57 : Refinement result formatting example

X.2 Implementing results

Once all variables, operations and initialisation have been successfully refined, Bart must produce output components and implement variables, operations and initialisation parts in these components.

Bart output splitting process is driven by operation refinement results and by variables used by those. Once the tool has decided how operations must be implemented along the output chain, variables and initialisations parts are dispatched according to operation arrangement.

X.2.1 Splitting operations in output components

For each operation to implement, Bart considers two sets of variables:

- *Variables to implement:* This set contains all variables present in IMPLEMENT clauses of substitution rules found for this operation. These are variables that must be implemented in the machine for the operation to be implemented
- *Exported variables:* These are abstract variables used in specifications of imported operations generated for this one refinement. These variables must not be implemented as long as the operation is not implemented

In the following, $\text{exported}(\text{op})$ are variables exported by operation op , and $\text{implement}(\text{op})$ are variables to implement for operation op . Term “before” and “further” refers to the order of the output chain

Bart chooses operations arrangement by generating iteratively output components with respect of following constraints:

- An operation must be implemented before imported operations defined for its refinement
- If the operation op is implemented in current component, other operations opX have to be implemented further if intersection of $\text{exported}(\text{op})$ and $\text{implement}(\text{opX})$ is not empty

X.2.2 Resolving deadlocks

X.2.2.1 Bart splitting algorithm

This section presents the algorithm used by Bart to split operations with respect for constraints exposed in X.2.1.

First, the set of operations to implement is filled with operations of original component to refine. Then Bart repeats following process as long as no error occurs and there are still operations to implement:

- The tool builds the set E containing variables exported by all operation that must be currently implemented.
- Each operation “ op ” such as intersection of $\text{implement}(\text{op})$ and E is empty is implemented in current component, and is removed of set of operations to implement
- Operation of the set that could not be implemented in current component will be promoted in the implementation
- Once every operation has been tried, imported operations eventually defined by refinement of the ones implemented in current component are added to the set of operations to implement
- If the set of operations to implement is not empty, process goes on with a new generated output component

Following figure shows some example operations and their generated imported operations:

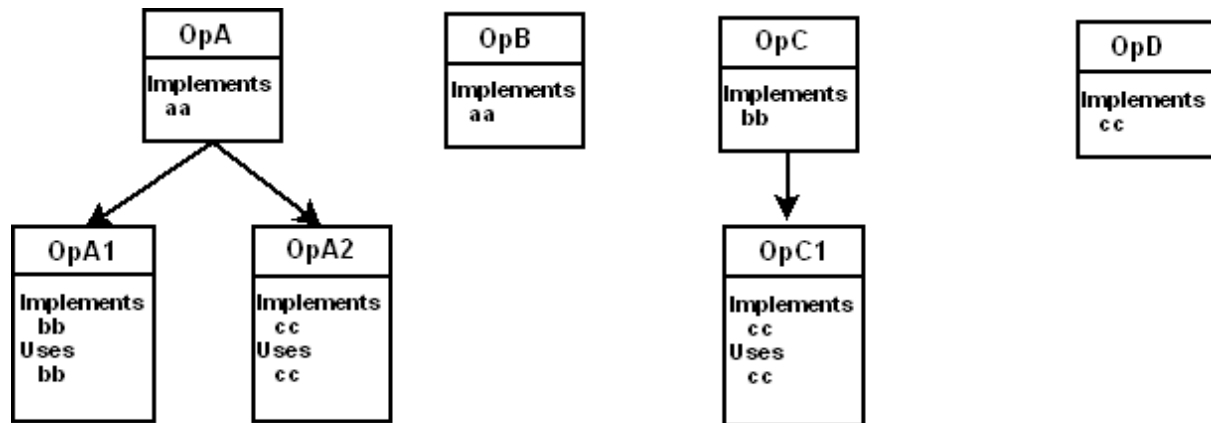


Figure 58 : Operations to implement for splitting example

For these operations, Bart may generate following machines:

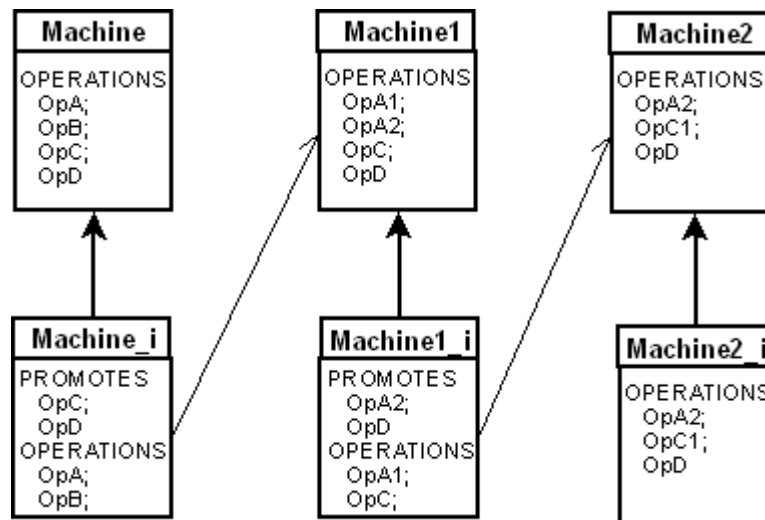


Figure 59 : Result machines for splitting example

The generation process is as follow:

- Step 1, operations to implement are {OpA, OpB, OpC, OpD}
 - Variables exported by all operations are {bb, cc}
 - OpA and OpB don't contain those in their variables to implement, they can be implemented
 - OpC and OpD can not be implemented, they will be promoted
- Step 2, operations to implement are {OpA1, OpA2, OpC, OpD}
 - Exported variables are {cc}

- OpA1 and OpC can be implemented
- OpA2 and OpD are promoted
- Step 3, operations to implement are {OpA2, OpC1, OpD}
 - There are no exported variables anymore, all operations can be implemented

Figure 60 : Splitting process example

X.2.2.2 What is a splitting deadlock?

A splitting deadlock is an error in the process previously described in X.2.2.1.

It occurs when, at a certain splitting step, no operation can be implemented by Bart in current component. It means that every operation has one of its variables to implement contained in another one exported variables.

For example, following draw shows a deadlock case:

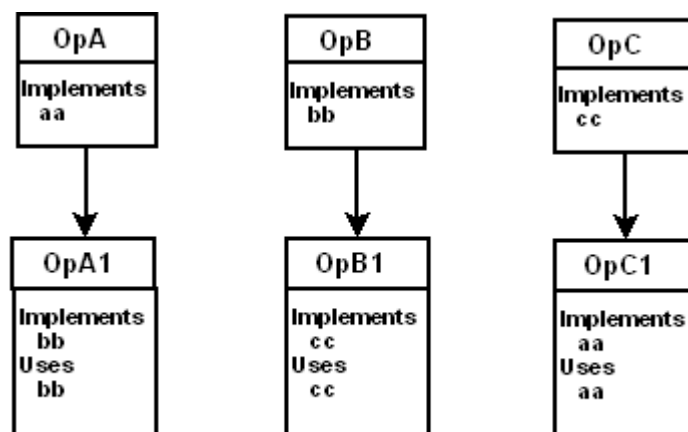


Figure 61 : Splitting deadlock example

Each operation to be implemented in current component needs another to be implemented further. So no operation can be implemented at current step and an error occurs.

X.2.2.3 Solving a deadlock case

When a deadlock occurs, Bart tries some processes to automatically solve it. It checks whether splitting conflicting operation bodies in several parts and

putting them in imported operations may solve the problem. If so, the result is generated and the operation is transparent to the user.

But in some cases, Bart is not able to solve automatically the problem. Then it generates a `deadlock.xml` file in the component directory. This file contains a XML description of the conflicting situation (operations, exported variables and variables to implement). It can be provided to the Bart GUI, which will display a draw representing the deadlock.

A deadlock is often caused by cycle as described in the example of X.2.2.2. In this case, the user should modify used rules to split more operation bodies and not have operations needing at the same time to implement and export variables.

XI APPENDIX A – FIGURES TABLE

Figure 1 : Bart command line parameters.....	7
Figure 2 : Example of Bart standard output.....	8
Figure 3 : Example of Bart verbose output mode.....	9
Figure 4 : Example of selection of operations to refine.....	10
Figure 5 : Refinement process order.....	11
Figure 6 : Examples of pattern-matching without previous instantiation.....	13
Figure 7 : Examples of @a + @b pattern-matching with previous instantiation.	13
Figure 8 : Hypothesis stack for constraint checking examples.....	14
Figure 9 : Examples of constraint checking.....	14
Figure 10 : Bart testing rule process.....	15
Figure 11 : Hypothesis stack filling with environment.....	16
Figure 12 : Example of Bart output components.....	17
Figure 13 : Bart expression guards.....	19
Figure 14 : Bart predicate guards.....	20
Figure 15 : Bart substitution guards.....	20
Figure 16 : Predicate theory example.....	21
Figure 17 : Substitution for EMPILE_PRE and DEPILE_PRE example.....	22
Figure 18 : Example of stack evolution with EMPILE_PRE and DEPILE_PRE.....	23
Figure 19 : Processing variable theories to find rules.....	28
Figure 20 : Searching variables refined by a particular rule.....	28
Figure 21 : Theories and stack for variable rule research example.....	29
Figure 22 : Variable rule research example.....	29
Figure 23 : Rules for type predicate example.....	30
Figure 24 : Example of type predicates adding.....	30
Figure 25 : Example of variable refinement rule with variable implementation.	31
Figure 26 : Variable refinement rule with concrete variable.....	32
Figure 27 : Variable refinement rule with abstract variable.....	33
Figure 28 : Theories and stack for operation rule research.....	36
Figure 29 : Example of operation rule research.....	36
Figure 30 : Process of substitution refinement.....	38
Figure 31 : Substitution and theories for rule tree example.....	39
Figure 32 : Example of refinement rule tree.....	40
Figure 33 : Bart predefined refinement behaviours.....	41
Figure 34 : Type iteration generated loop, without while parameter.....	43
Figure 35 : Type iteration generated loop, with while parameter.....	44
Figure 36 : Type iteration generated machine.....	44
Figure 37 : Invariant iteration generated loop.....	45
Figure 38 : Invariant iteration generated machine.....	46
Figure 39 : Concrete iteration generated loop.....	47
Figure 40 : Example of generated iterators.....	48
Figure 41 : Imported operation naming example.....	50

Figure 42 : Simple imported operation example.....	50
Figure 43 : Imported operation example with parameters adding.....	51
Figure 44 : Example of imported operation precondition adding.....	52
Figure 45 : Substitution and rule for LH example.....	53
Figure 46 : Example of stack filling with LH substitutions.....	54
Figure 47 : Local variable declaration example.....	55
Figure 48 : Usage of structure and operation rules.....	57
Figure 49 : Theories and operation for operation refinement example.....	58
Figure 50 : Operation refinement example.....	59
Figure 51 : Substitution and theories for initialisation refinement example.....	60
Figure 52 : Initialisation refinement example.....	61
Figure 53 : User pass theory example.....	62
Figure 54 : Tactic theory example.....	64
Figure 55 : Usage of tactics and user passes.....	64
Figure 56 : Bart refinement result formatting.....	67
Figure 57 : Refinement result formatting example.....	67
Figure 58 : Operations to implement for splitting example.....	69
Figure 59 : Result machines for splitting example.....	69
Figure 60 : Splitting process example.....	70
Figure 61 : Splitting deadlock example.....	70

XII APPENDIX B – SYNTAX ELEMENTS TABLE

Syntax 1 : Predicate theory.....	21
Syntax 2 : Rule files.....	25
Syntax 3 : Variable rule theories.....	27
Syntax 4 : Operation rule theories.....	35
Syntax 5 : Type iteration.....	43
Syntax 6 : Invariant iteration syntax.....	45
Syntax 7 : Concrete iteration.....	47
Syntax 8 : Seen operation.....	48
Syntax 9 : Imported operation.....	49
Syntax 10 : Implement.....	52
Syntax 11 : LH.....	53
Syntax 12 : Structure theories.....	56
Syntax 13 : Initialisation theories.....	59
Syntax 14 : User pass theory.....	62
Syntax 15 : Tactic theory.....	63

XIII APPENDIX C – RULE FILES COMPLETE SYNTAX

This section presents the complete Bart rule files syntax.

XIII.1 Rule files

RuleFile = [Theory { "&" Theory }].

```

Theory
=
    VariableTheory
    | OperationTheory
    | StructureTheory
    | InitialisationTheory
    | UserPassTheory
    | TacticTheory
    | PredicateTheory
.

```

XIII.2 Variables refinement rules

```

VariableTheory
=
    "THEORY_VARIABLE" ident
    "IS"
    VariableRule { ";" VariableRule }
    "END" ident
.

VariableRule
=
    "RULE" ident
    ["(" JokerList ")"]
    "VARIABLE" JokerList
    [ "TYPE" ident "(" JokerList ")" ]
    [ "WHEN" Predicate ]
    "IMPORT_TYPE" Predicate
    (VariableImplementation | VariablesRefinement )
    "END"
.

VariableImplementation =
    "CONCRETE_VARIABLES" JokerList
    [ "DECLARATION" Predicate ]
    "INVARIANT" Predicate
.

VariablesRefinement =
    "REFINEMENT_VARIABLES"
    VariableRefinement { "," VariableRefinement }
    "GLUING_INVARIANT" Predicate
.

```

```

.
VariableRefinement =
  "CONCRETE_VARIABLE" joker
  "WITH_INV" Predicate
  "END"
|
  "ABSTRACT_VARIABLE" joker
  "REFINED_BY" ident "." ident "(" Expression ")"
  "WITH_INV" Predicate
  "END"
.

```

XIII.3 Initialisation refinement rules

```

InitialisationTheory
=
  "THEORY_INITIALISATION" ident
  "IS"
    InitialisationRule { ";" InitialisationRule }
  "END" ident
.

InitialisationRule
=
  "RULE" ident
  "REFINES" Substitution
  [ "WHEN" Predicate ]
  "IMPLEMENTATION" Substitution
  "END"
.

```

XIII.4 Operation refinement rules

```

OperationTheory
=
  "THEORY_OPERATION" ident
  "IS"
    OperationRule { ";" OperationRule }
  "END" ident
.

OperationRule
=
  "RULE" ident
  "REFINES" Substitution
  [ "WHEN" Predicate ]
  [ "SUB_REFINEMENT" SubRefinementRule { "," SubRefinementRule } ]
  ( "REFINEMENT" | "IMPLEMENTATION" )
    { RefinementVarDecl }
  Substitution
  [ "IMPLEMENT" IdentOrJokerList ]
  "END"
.

RefinementVarDecl =

```

```

("VARIABLE" | "ABSTRACT_VARIABLE" | "CONCRETE_VARIABLE") joker
[ "REFINED_BY" ident "(" Expression ")" ]
"WITH_INV" Predicate
"WITH_INIT" Substitution
"IN"
.

SubRefinementRule =
  "(" Substitution ")" "->" "(" Substitution ")"
.

```

XIII.5 Structural refinement rules

```

StructureTheory
=
  "THEORY_STRUCTURE" ident
  "IS"
    OperationRule { ";" OperationRule }
  "END" ident
.

```

XIII.6 User pass theory

```

UserPassTheory
=
  "USER_PASS" "
  IS"
  [ ("VARIABLE"|"OPERATION"|"INITIALISATION") ":" "(" IdentList ")" ]
  { ";" ("VARIABLE"|"OPERATION"|"INITIALISATION") ":" "(" IdentList ")" }
  "END"
.

```

XIII.7 Tactic theory

```

TacticTheory
=
  "THEORY" "TACTICS" "IS"
    { Tactic }
  "END"
.

Tactic =
  "VARIABLE" ":" VariableTactic { ";" VariableTactic }
|
  "INITIALISATION" ":" SubstitutionTactic { ";" SubstitutionTactic }
|
  "OPERATION" ":" SubstitutionTactic { ";" SubstitutionTactic }
.

SubstitutionTactic =
  IdentList "=>" "(" Substitution ")"
.

VariableTactic =
  IdentList "=>" "(" Predicate ")"

```

XIII.8 Predicate synonyms theory

```

PredicateTheory
=
    "THEORY_PREDICATES"
    "IS"
        PredicateDefinition { "|" PredicateDefinition}
    "END"
.

PredicateDefinition
=
    ident "(" JokerList ")" "<=>" Predicate
.

```

XIII.9 Substitutions

```

Substitution = SimpleSubstitution { ("||";") SimpleSubstitution }.

SimpleSubstitution
=
    "skip"
|   "BEGIN" Substitution "END"
|   "PRE" Predicate
|   "THEN" Substitution
|   "END"
|   "ASSERT" Predicate
|   "THEN" Substitution
|   "END"
|   "CHOICE" Substitution
|   { "OR" Substitution }
|   "END"
|   "IF" Predicate
|   "THEN" Substitution
|   { "ELSIF" Predicate
|   "THEN" Substitution }
|   [ "ELSE" Substitution ]
|   "END"
|   "SELECT" SelectContent [ "ELSE" Substitution ] "END"
|   "CASE" Expression "OF"
|   "EITHER" PrimaryExpression
|   "THEN" Substitution
|   { "OR" PrimaryExpression
|   "THEN" Substitution }
|   [ "ELSE" Substitution ]
|   "END"
|   "ANY" IdentOrJokerList
|   "WHERE" Predicate
|   "THEN" Substitution
|   "END"
|   "LET" IdentOrJokerList "BE"
|   Predicate
|   "IN" Substitution

```

```

    "END"
  |   "VAR" IdentOrJokerList
    "IN" Substitution
    "END"
  |   "WHILE" Predicate
    "DO" Substitution
    "INVARIANT" Predicate
    "VARIANT" Expression
    "END"
  |   "LH" Predicate
    "THEN" Substitution
    "END"
  |   joker
  |   ident
  |   AffectSubstitution
  |   Iteration
  |   ImportedOperation
  |   ImplementSubstitution
  .

SelectContent =
  Predicate
  "THEN" Substitution
  { "WHEN" Predicate
    "THEN" Substitution }
  .

ImplementSubstitution = "IMPLEMENT" "(" Substitution ")".

ImportedOperation
=
  "IMPORTED_OPERATION" "("
  [ "name" "=>" ident "," ]
  "out" "=>" "(" [ IdentJokerVardeclList ] ")" ","
  "in" "=>" "(" [ IdentJokerVardeclList ] ")" ","
  "pre" "=>" "(" Predicate ")" ","
  "body" "=>" "(" Substitution ")"
  ")"
  |
  "SEEN_OPERATION" "("
  "name" "=>" IdentOrJoker ","
  "out" "=>" "(" [ IdentJokerVardeclList ] ")" ","
  "in" "=>" "(" [ IdentJokerVardeclList ] ")" ","
  "body" "=>" "(" Substitution ")"
  ")"
  .

Iteration
=
  "INVARIANT_ITERATION" "("
  [ ("tant_que"|"while") "=>" IdentOrJokerOrVarDecl "," ]
  "1st" "index" "=>" BinaryExpression115 ","
  "2nd" "index" "=>" BinaryExpression115 ","
  "constant" "=>" BinaryExpression115 ","
  "1st" "type" "=>" BinaryExpression115 ","
  "2nd" "type" "=>" BinaryExpression115 ","
  "body" "=>" "(" Substitution ")" ","
  "invariant" "=>" "(" Predicate ")"
  ")"
  |
  "TYPE_ITERATION" "("
  [ ("tant_que"|"while") "=>" IdentOrJokerOrVarDecl "," ]

```

```

    "index" "=>" BinaryExpression115 ","
    "type" "=>" BinaryExpression115 ","
    "body" "=>" "(" Substitution ")" ","
    "invariant" "=>" Predicate
  )"
|
  "CONCRETE_ITERATION" "("
  "init_while" "=>" "(" Substitution<out Substitution init> ")" ","
  ("tant_que"|"while") "=>" BinaryExpression115<out Expression e> ","
  "body" "=>" "(" Substitution<out Substitution body> ")" ","
  "invariant" "=>" "(" Predicate<out Predicate invariant> ")" ","
  "variant" "=>" BinaryExpression115<out Expression variant> ","
  "flag" "=>" IdentOrJoker<out Expression flag>
  ")"
.

AffectSubstitution
=
  IdentJokerVardeclList
  ( ":" "(" Predicate ")"
  | "::" Expression
  | [ "(" Expression ")" ]
  ":" Expression
  | "<--" IdentOrJoker [ "(" Expression ")" ]
  )
.

```

XIII.10 Predicates

```

Predicate = ConjunctionPredicates { "=>" ConjunctionPredicates }.

ConjunctionPredicates
=
  EquivalencePredicate { ("&"|"or"|"cand") EquivalencePredicate }
.

EquivalencePredicate = SimplePredicate { "<=>" SimplePredicate }.

SimplePredicate
=
  "(" Predicate ")"
  | "bnot" "(" Predicate ")"
  | joker
  | ident "(" Expression ")"
  | Expression ComparisonOperator Expression
  | "not" "(" Predicate ")"
  | ("!"|"#") QuantifiedList "." "(" Predicate ")"
.

ComparisonOperator =
  "=" | "/" = | ":" | "/" : | "<:" | "<<:" | "<:" | "<<:" | "<=" | ">=" | ">" | "<"
.

```

XIII.11 Expressions

Expression = BinaryExpression20.

// Expression that can occur between operators of priority 20

BinaryExpression20 = BinaryExpression115 { "," BinaryExpression115 }.

// Expression that can occur between operators of priority 115

BinaryExpression115 =
 BinaryExpression125 { Operator115 BinaryExpression125 }

.

Operator115 = "<->" | "+->" | "+->>" | ">->" | "->" | ">+>>" | ">+>" | "->>>" | ">->>".

// Expression that can be located between operators of priority 125

BinaryExpression125 =
 BinaryExpression160 { Operator125 BinaryExpression160 }

.

Operator125 =

"<-" | "><" | "<<" | ">>" | "<" | ">" | "\" | "/" | "^" | "->" | "\" | "/" | "<+" | "/" | "\" | ">" | "->"

.

// Expression that can occur between operators of priority 160

BinaryExpression160
 =
 BinaryExpression170 { ".." BinaryExpression170 }

.

BinaryExpression170

=
 BinaryExpression180 { ("+" | "-") BinaryExpression180 }

.

BinaryExpression180

=
 BinaryExpression190 { ("*" | "/" | "mod") BinaryExpression190 }

.

BinaryExpression190

=
 Expression200 { "***" Expression200 }

.

Expression200

=
 Expression210
 |
 "-" Expression210

.

Expression210

=
 PrimaryExpression
 {
 "~"
 |
 ("[" Expression "]")
 |
 ("(" Expression ")")
 }
 }

.

```

PrimaryExpression
=
    ident
  |   number
  |   joker
  |   vardecl
  |   "(" Expression { (";"|"|"|"") Expression } ")"
  |   "MAXINT"
  |   "MININT"
  |   "{}"
  |   "[]"
  |   FuncOperator "(" Expression ")"
  |   "{" Expression [ "|" Predicate ] "}"
  |   "[" Expression "]"
  |   "TRUE"
  |   "FALSE"
  |   "bool" "(" Predicate ")"
  |   "%" QuantifiedList "." "(" Predicate "|" Expression ")"
.

QuantifiedList
=
    IdentOrJokerList
  |   "(" IdentOrJokerList ")"
.

FuncOperator =
    "max"|"min"|"card"|"dom"|"ran"|"POW"|"POW1"|"FIN"|"FIN1"|"union"|"inter"
.

```

XIII.12 Diverse

```

JokerList = joker { "," joker }.

IdentOrJokerList = IdentOrJoker { "," IdentOrJoker }.

IdentOrJoker = ident | joker.

IdentJokerVardeclList =
    IdentOrJokerOrVardecl { "," IdentOrJokerOrVardecl }
.

IdentOrJokerOrVardecl = ident | joker | vardecl.

IdentList = ident { "," ident }.

```