



Edition	1
Revision	7
Page number	110
State	To be checked

BART **VERSION 1.2** - **USER MANUAL**

	Redaction	Checked	Approved
Function	Developer		
Name	Mathieu VIALE		
Date	21/08/2014		
Signature			

Réf. :	Version : 1.6	Date : 05/06/2015
--------	---------------	-------------------

REVISIONS

Version	Date	Author	Comment
1.0		M. Viale	Initial version
1.1	17/10/2013	M. Viale	For Bart 1.1: typing information, substitution guards, merging imported operations.
1.2	03/04/2014	M. Viale	Add of information about use of TYPE_ITERATION Quality modification
1.3	20/06/2014	F. Deschamps	Add accessor functioning description
1.4	22/07/2014	F. Deschamps	Add EXPLICIT_REFINEMENT clause description. Update parts concerning variables refinement, variables declaration in operation rules and refinement components generation and add some examples. Description of bfree guards
1.5	21/08/2014	F. Deschamps	Update clause REFINED_BY of variable and operation rules. Add two new special predefined refinement behaviours
1.6	05/06/2015	F. Deschamps	Add explanations about computation of well-definedness conditions and typing predicates
1.7	01/11/2018	D. Deharbe	Mise à jour des paramètres de l'exécutable.

INDEX

Revisions.....	2
Index.....	3
I Introduction.....	6
II Usage.....	7
II.1 Command usage	7
II.2 Input files	8
II.3 Visibility for loaded components	8
II.4 Bart standard output verbosity	9
II.5 Bart rule trace	9
III Automatic refinement principles.....	11
III.1 Refined elements.....	11
III.1.1 Abstract variables	11
III.1.2 Operations.....	11
III.1.3 Initialisation.....	12
III.1.4 Process	12
III.2 Pattern-Matching.....	13
III.2.1 Jokers syntax.....	13
III.2.2 Pattern matching	13
III.3 Refinement rules	14
III.3.1 Introduction.....	14
III.3.2 Constraints	14
III.3.3 Guards	15
III.3.4 Rule checking process	16
III.3.5 Jokers use in result.....	17
III.4 Hypothesis stack – Environment analysis.....	17
III.5 Result production and writing.....	18
IV Bart guards – Predicate synonyms	20
IV.1 Guards.....	20
IV.1.1 Expression guards	20
IV.1.2 Predicate guards.....	21
IV.1.3 Substitution guards.....	22
IV.1.4 Heterogeneous guards	22
IV.2 Predicate synonyms.....	23
V Pragmas and comments.....	25
V.1 EMPILE_PRE, DEPILE_PRE	25
V.2 Magic.....	26
V.2.1 For variables	26
V.2.2 For substitutions.....	26

V.3	CAND	26
VI	Rule files.....	28
VI.1	Syntax.....	28
VI.2	Using rule files.....	29
VI.2.1	Providing rule files on command line	29
VI.2.2	Rule file associated to the component	29
VI.2.3	Bart refinement rule base	29
VII	Variables refinement	30
VII.1	Variable theories syntax	30
VII.2	Variable rule research	31
VII.3	Storing information predicates about found variable rules.....	33
VII.4	Invariant for refined abstract variables.....	34
VII.5	Specifying variable refinement results.....	34
VII.5.1	Using CONCRETE_VARIABLES clause.....	34
VII.5.2	Using REFINEMENT_VARIABLES clause	35
VIII	Substitution refinement	38
VIII.1	Rule syntax	38
VIII.2	Rule research.....	39
VIII.3	Refinement process	40
VIII.4	Default refinement behaviours	45
VIII.4.1	Basic behaviours.....	45
VIII.4.2	Special behaviours	46
VIII.5	Special refinement substitutions	50
VIII.5.1	Iterators	51
VIII.5.2	Using operations from seen machines - SEEN_OPERATION.....	57
VIII.5.3	Defining imported operations - IMPORTED_OPERATION	58
VIII.5.4	Controlling the refinement process.....	62
VIII.5.5	Local variable declarations	64
VIII.6	Declaring operation refinement variables.....	65
VIII.7	Usage of substitution rules	68
VIII.7.1	Accessor rules	68
VIII.7.2	Structural and operation rules - Operation refinement	76
VIII.7.3	Initialisation rules	79
IX	Tactic and user pass theories.....	82
IX.1	User pass theory	82
IX.1.1	Syntax	82
IX.1.2	Usage	82
IX.2	Tactic theory	83
IX.2.1	Syntax	83
IX.2.2	Usage	83
IX.3	Priority of Tactic and User pass theories	84
X	Result production and writing	86
X.1	Formatting the result.....	86
X.2	Well-definedness conditions and typing predicates	87
X.3	Generating refinement components.....	88

X.4	Implementing results.....	92
X.4.1	Splitting operations in output components	93
X.4.2	Resolving deadlocks.....	93
XI	Appendix A – Figures table.....	97
XII	Appendix B – Syntax elements table.....	99
XIII	Appendix C - Configuring Bart in AtelierB with resources.....	100
XIV	Appendix D – Rule files complete syntax.....	101
XIV.1	Rule files	101
XIV.2	Variables refinement rules	101
XIV.3	Initialisation refinement rules.....	102
XIV.4	Operation refinement rules	102
XIV.5	Accessor refinement rules.....	103
XIV.6	Structural refinement rules	103
XIV.7	User pass theory	104
XIV.8	Tactic theory.....	104
XIV.9	Predicate synonyms theory.....	104
XIV.10	Substitutions.....	105
XIV.11	Predicates.....	107
XIV.12	Expressions	107
XIV.13	Diverse	109

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 interfaces (bbatch and GUI).

II USAGE

II.1 Command usage

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

```
Bart [options] { -r rule_file }* -m machine_file
```

Options:

Parameter	Comment
-h	Displays help
-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
-E encoding	File encoding
-v	Displays more information
-V	Displays more information than -v
-f	Forces BART to produce output even in case of failure
-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 Atelier Database Repository (stores desc files)
-R resource	Loads the specified resource file
-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
-A dir	Stores generated XML files in directory dir
-l	Displays guards list
-F name	Use given resolving information for finding path of given component file
-P path	Path where ML.kin can be found, if it is not in the same directory as the bart executable (used by the guard PR)
-T param	Add extra typing information to the generated code (required by some translators) Possible values for param: PRE : add type information for output parameters in preconditions VAR : add type information for local variables in VAR IN substitutions Multiple -T options can be used.

-k params	sets the memory parameters to use when calling the prover (for PR guards). In the case where a resource file is provided, the value from the resource file is used
-i	Gives current Bart version
-O	Merges identical generated imported operations
-W	Adds well-definedness conditions to imported operation preconditions
-W	Adds typing predicates to while invariants

Figure 1 : Bart command line parameters

Deprecated options:

Option `-b` (BDP path of the project) is no longer supported. Instead BART reads the DESC file of the project in Atelier Database Repository (bdb directory).

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 loads seen machines or the abstraction(s) of the given component, it must be able to find their associated files on the file system. Two search modes are available: path-based and Atelier B-based. These search modes are mutually exclusive.

To use path-based search, use the following parameters:

- `-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

To use Atelier B-based, use the following parameters:

- `-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 workspace bdb path. Parameters -B and -p must be present together.

Again, these two modes cannot be mixed.

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:

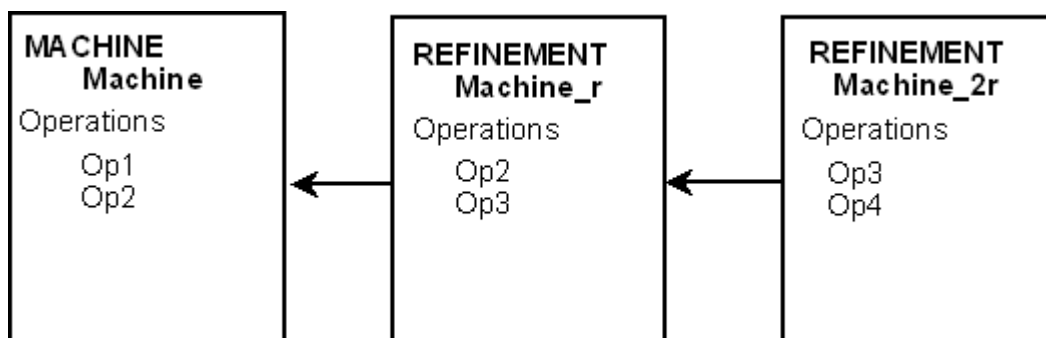


Figure 4 : Example of selection of operations to refine

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.

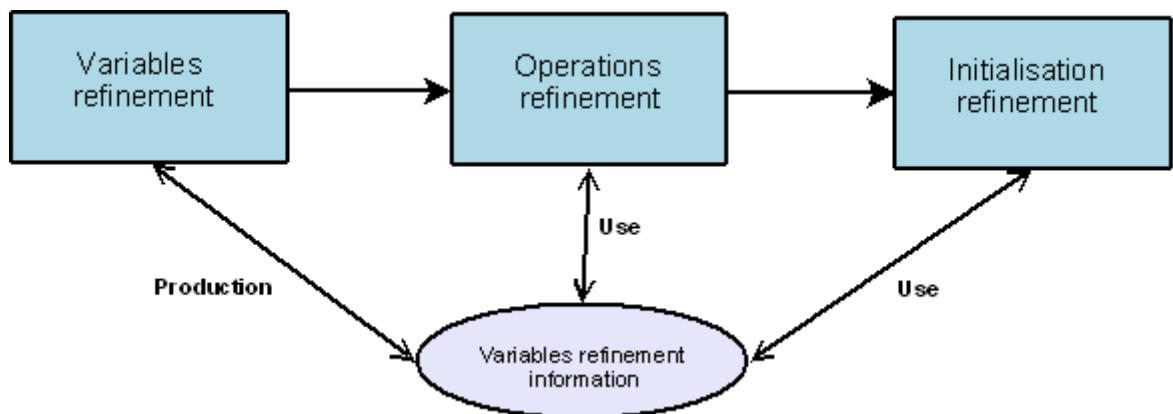


Figure 5 : Refinement process order

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 va1 THEN aa := 0 ELSE aa := 1 END	Success	{@a = IF va1 THEN aa := 0 ELSE aa := 1 END }
IF @p THEN	IF va1 THEN	Success	{@p = va1, @t = aa :=0, @e =

@t ELSE @e END	aa := 0 ELSE aa := 1 END		aa := 1}
IF @_ THEN @t ELSE @e END	IF va1 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: $@a \leq @b \ \& \ ABVAR(@b) \ \& \ bnum(@a)$, with $@b$ instantiated.

It checks that $@b$ is an abstract variable (predicate present in the stack), that a predicate $@a \leq @b$ 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:

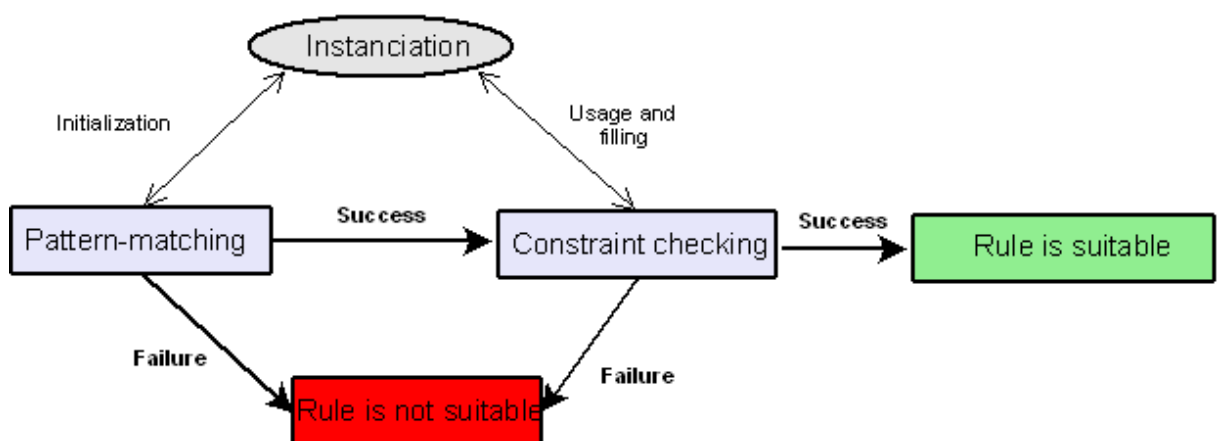


Figure 10 : Bart testing rule process

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":

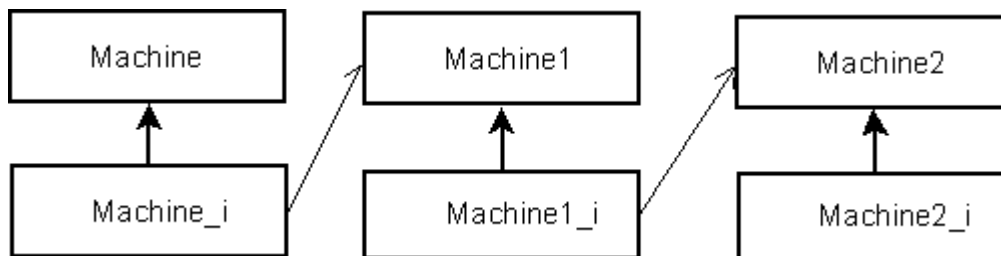


Figure 12 : Example of Bart output components

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.

Bart may add some refinement components between the machines and their implementations if it is necessary, to declare new abstract variables for example.

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	<p>"joker" must be a single joker. This guard makes joker match with expr. Joker may be uninstantiated.</p> <p>If joker is not instantiated, the guard is true and joker value is now expr</p> <p>If joker is instantiated, guard is true if g can match with joker instantiation.</p>
PAR_IN(expr)	Stack	<p>Checks if given expression is an identifier of a currently refined operation input parameter.</p> <p>These guards are added to the stack when a new operation refinement begins</p>
PAR_OUT(expr)	Stack	<p>Checks if given expression is an identifier of a currently refined operation output parameter.</p> <p>These guards are added to the stack when a new operation refinement begins</p>
REFINED(expr)	Stack	<p>Checks if given expression is an identifier of a variable introduced by another variable refinement.</p> <p>These guards are added after variables refinement phase</p>
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	<p>Checks if given parameter is a concrete variable introduced by the operation refinement process or by variables refinement.</p> <p>Added when the concrete variable is introduced</p>
VAR_LOC(expr)	Stack	<p>Checks if given parameter is a local variable introduced by current operation refinement.</p> <p>Added when the local variable is introduced</p>

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	<p>"oper" must be an operation description that can contain jokers.</p> <p>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.</p> <p>A guard of this type is added for each operation of machines seen by the component to refine.</p>
bhasflow(sub)	Calculated	Checks if given substitution contains flow (i.e. branch structures as IF or SELECT substitutions)
bsearch(pattern list result)	Calculated	<p>Checks if <i>pattern</i> substitution is present in <i>list</i> substitution.</p> <p>If so, <i>result</i>, which must be an uninstantiated joker, takes the value of list without pattern occurrences.</p>

Figure 15 : Bart substitution guards

IV.1.4 Heterogeneous guards

Heterogeneous guards all called so because they have several parameters which may have different types (expressions, substitutions or predicates).

Parameters of heterogeneous guards are always separated by “|” symbol.

Guard	Type	Description
bsub(sub1 sub 2 exp1 exp2)	Calculated	Replaces expression exp1 by expression exp2 in the substitution sub1. The guard returns true if replacing result matches sub2 (even if no replacement was performed). exp1 and exp2 may contain uninstantiated joker. sub2 is usually an uninstantiated joker but may be a more complex substitution.
bsubpred(pre1 pre2 exp1 exp2)	Calculated	Replaces expression exp1 by expression exp2 in the predicate pre1. The guard returns true if replacing result matches pre2 (even if no replacement was performed). exp1 and exp2 may contain uninstantiated joker. pre2 is usually an uninstantiated joker but may be a more complex substitution.
bsubexpr(exp3 exp4 exp1 exp2)	Calculated	Replaces expression exp1 by expression exp2 in the expression exp3. The guard returns true if replacing result matches exp4 (even if no replacement was performed). exp1 and exp2 may contain uninstantiated joker. exp4 is usually an uninstantiated joker but may be a more complex substitution.
bfree(list sub)	Calculated	Checks if identifiers of list are free in substitution sub
bfreepred(list pred)	Calculated	Checks if identifiers of list are free in predicate pred
bfreeexpr(list expr)	Calculated	Checks if identifiers of list are free in expression expr

Figure 16 : Bart heterogeneous 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 17 : 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 18 : 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 &	valeur > 0 &

	Previous predicates	valeur > 100 & Previous predicates
Else branch	Not(valeur > 100) & Previous predicates	Previous predicates

Figure 19 : 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, accessor, 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
```

```
| AccessorTheory
```

```
| 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" JokerList
	["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 20 : 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 21 : 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
<pre> THEORY_VARIABLE t1 IS 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> aa : INT & bb : INT & cc : NAT & dd : NAT & ee : NAT & value < aa & bb < 0 & 0 <= cc & 1 <= dd & 1 <= ee & ABCON(value) </pre>

Figure 22 : 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 23 : 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 24 : 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 25 : 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, @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 26 : 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. Jokers contained in subparts of these elements must all have been previously instantiated. For refinement abstract variables, you can provide in REFINED_BY clause a theory name or directly a rule with its parameters (cf. example) that will be used to refine it. It is possible to define several abstract variables at once so you can specify a unique rule to refine them all. If REFINED_BY clause is not present, the new variables will be refined as original ones by processing the rule files.

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, @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 27 : Variable refinement rule with concrete variable

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

```

RULE r_sequence
VARIABLE
  @a
TYPE
  sequence(@a,@s,@m)
WHEN
  @a : seq(@s) &
  (SET(@s) or COCON(@s)) &
  COCON(@m) & size(@a)<=@m
IMPORT_TYPE
  @a : seq(@s)
REFINEMENT_VARIABLES
  ABSTRACT_VARIABLE
    @a_r
  REFINED_BY
    my_var.r_fonc(@a_r)
  WITH_INV
    @a_r : 0..@m +-> @s
  END,
  CONCRETE_VARIABLE
    @a_size_i
  WITH_INV
    @a_size_i : NAT
  END
GLUING_INVARIANT
  @a = 1..@a_size_i <| @a_r &
  @a_size_i = size(@a)
END

```

Figure 28 : Variable refinement rule with abstract variable

Here we directly specify which rule will be used to refine the new abstract 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 r_fonc(@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, accessor, initialisation and structural rules. Operation and structure rules are identical. Accessor rules are a special kind of operation rules which encapsulate their result clause into an imported operation. 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" | "EXPLICIT_REFINEMENT" | "IMPLEMENTATION" )
    { RefinementVarDecl }
    Substitution
    ["IMPLEMENT" IdentOrJokerList ]
    "END"
.

```

```

RefinementVarDecl =
(
  ("VARIABLE" | "ABSTRACT_VARIABLE") VarDeclList
  [ "REFINED_BY" ident [ "." ident "(" Expression ")" ] ]
|
  "CONCRETE_VARIABLE" vardecl )
  "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 accessor theories of all rule files and then operation theories of all rule files as long as it could not find a rule.
- For each rule file it processes accessor or 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 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 error may occur (cf. VIII.3).

Each accessor and 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
<pre> 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) [...] END; RULE r_assign_plus_const REFINES @a := @b + @c WHEN PAR_OUT(@a) & ABCON(@b) & ABCON(@c) [...] END END assign_plus & THEORY_OPERATION assign_minus IS RULE assign_minus_1 REFINES @a := @b - @c [...] END END assign_minus </pre>	<pre> PAR_OUT(par_out) & PAR_IN(par_in1) & PAR_IN(par_in2) </pre>

Figure 29 : 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 30 : 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.

It is highly recommended not to put sub-refinements in a REFINEMENT rule. It could cause problems for the generation of intermediate refinements.

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.

EXPLICIT_REFINEMENT clause is the same as REFINEMENT one but it is used to indicate that the given result must appear in an intermediate refinement component. In that case, substitutions forbidden in refinements, as WHILE substitutions and IMPORTED_OPERATION, are naturally prohibited in the result clause.

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:

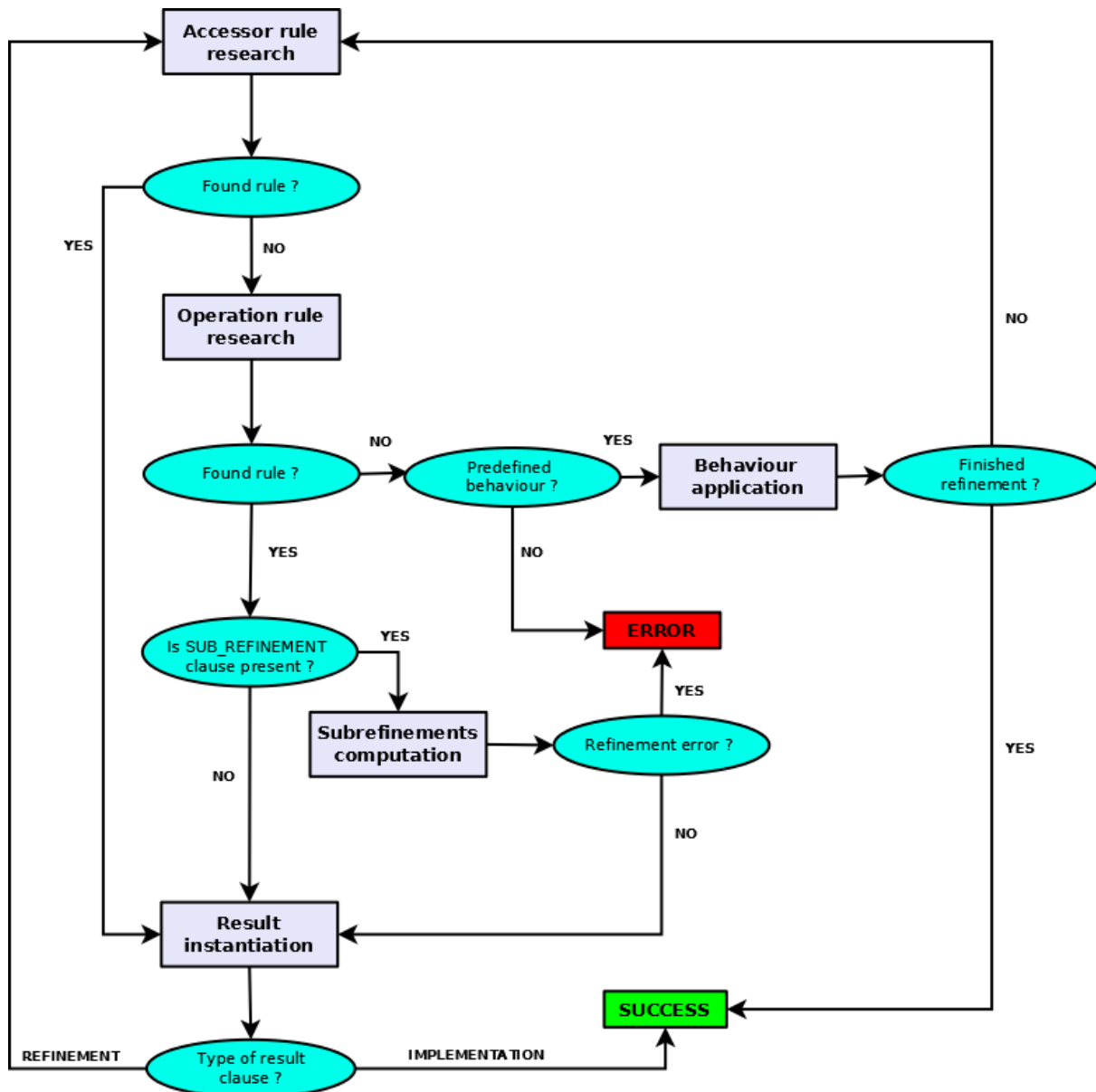


Figure 31 : Process of substitution refinement

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 ;

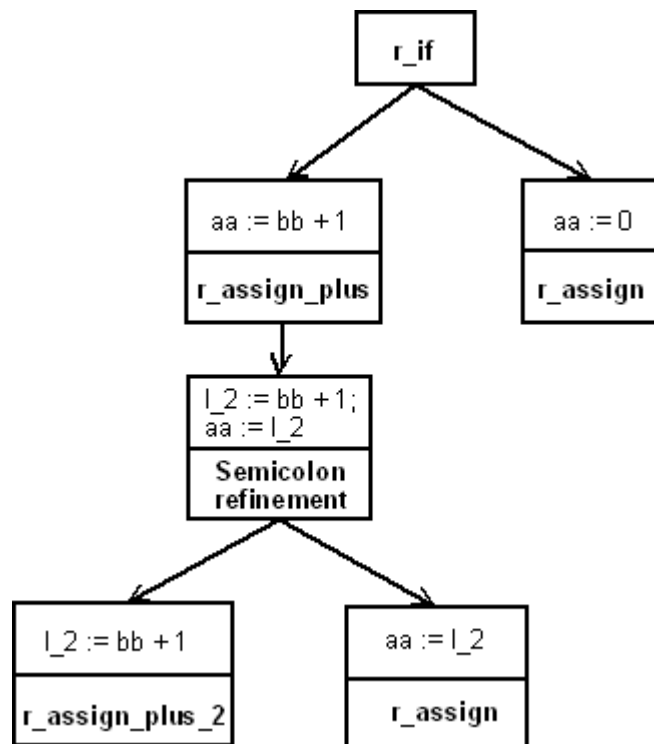
  RULE r_if
  REFINES
    IF @a THEN @b ELSE @c END
  SUB_REFINEMENT
    (@b) -> (@d),
    (@c) -> (@e)
  IMPLEMENTATION
    #1 := bool(@a);
    IF #1 = TRUE THEN @d ELSE @e END
  END
```

END theory

Figure 32 : 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 33 : 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

VIII.4.1 Basic 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	result(sequentialization(sub1; 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	ASSERT predicate THEN result(sub)	"predicate" is added to the hypothesis stack for refining "sub"

sub END	END	
Operation call	Refined by itself	Only if parameters are B0 expressions
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 34 : 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 $I_1 := aa ; aa := bb ; cc := I_1$, where I_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). If the variable modified and accessed (aa in the example) has not a simple B0 type (INT, BOOL, a set or an enum) Bart produces an error as it is not correct to affect an abstract expression to a local variable.

VIII.4.2 Special behaviours

The special predefined behaviours described below are applied only if no rule has matched and no basic behaviours can be used.

VIII.4.2.1 B0 substitutions

If the substitution to refine is detected as B0, it will be implemented directly by itself. The grammar used to determine if a substitution is B0 or not is based on the *B Language Reference Manual, section 7.25 Specificities of the B0 Language* and is described below:

Terms

Term ::=

- Simple_term*
- | *Arithmetic_expression*

Simple_term ::=

- Ren_ident*
- | *Lit_integer*
- | *Lit_boolean*
- | "bool(" *Condition* ")"

Ren_ident ::=

- Ident*

Lit_integer ::=

- Literal_integer*
- | "MAXINT"
- | "MININT"

Lit_boolean ::=

- "FALSE"
- | "TRUE"

Arithmetic_expression ::=

- Lit_integer*
- | *Ren_ident*
- | *Ren_ident* "(" *Term* "+", "*Term* ")"
- | *Arithmetic_expression* "+" *Arithmetic_expression*
- | *Arithmetic_expression* "-" *Arithmetic_expression*
- | "-" *Arithmetic_expression*
- | *Arithmetic_expression* "x" *Arithmetic_expression*
- | *Arithmetic_expression* "/" *Arithmetic_expression*
- | *Arithmetic_expression* "mod" *Arithmetic_expression*
- | *Arithmetic_expression* exp *Arithmetic_expression*
- | "succ" "(" *Arithmetic_expression* ")"
- | "pred" "(" *Arithmetic_expression* ")"
- | "(" *Arithmetic_expression* ")"

Expr_array ::=

- Ident*
- | "{" (*Simple_term* "+" | ">" | "<" | ">" *Term*) "+", "}"
- | *Range* "+" "x" "x" "{" *Term* "}"

Range ::=

- Ident*
- | *Interval_B0*
- | *Number_set_B0*
- | "BOOL"

Interval_B0 ::=
 Arithmetical_expression ".." *Arithmetical_expression*
 |
 Number_set_B0

Number_set_B0 ::=
 "NAT"
 |
 "NAT₁"
 |
 "INT"

Conditions

Condition ::=
 Simple_term "=" *Simple_term*
 |
 Simple_term "≠" *Simple_term*
 |
 Simple_term "<" *Simple_term*
 |
 Simple_term ">" *Simple_term*
 |
 Simple_term "≤" *Simple_term*
 |
 Simple_term "≥" *Simple_term*
 |
 Condition "∧" *Condition*
 |
 Condition "∨" *Condition*
 |
 "¬" "(" *Condition* ")"
 |
 "(" *Condition* ")"

Instructions

Instruction ::=
 Block_instruction
 |
 Local_variable_instruction
 |
 Identity_substitution
 |
 Becomes_equal_to_instruction
 |
 Operation_call_instruction
 |
 Conditional_instruction
 |
 Case_instruction
 |
 Assertion_instruction
 |
 Sequence_instruction
 |
 While_instruction

Block_instruction ::=
 "BEGIN" *Instruction* "END"

Local_variable_instruction ::=
 "VAR" *Ident* "+", "IN" *Instruction* "END"

Identity_substitution ::=
 "skip"

Becomes_equal_to_instruction ::=
 Ren_ident ["(" *Term* "+", ")"] ":" "=" *Term*
 |
 Ren_ident ":" "=" *Expr_array*

Operation_call_instruction ::=
 [*Ren_ident* "+", "←"] *Ren_ident* ["(*Term* +", ")"]

Sequence_instruction ::=
Instruction ",", *Instruction*

Conditional_instruction ::=
 "IF" *Condition* "THEN" *Instruction*
 ("ELSIF" *Condition* "THEN" *Instruction*) +
 ("ELSE" *Instruction*)
 "END"

Case_instruction ::=
 "CASE" *Simple_term* "OF"
 "EITHER" *Simple_term* +", " "THEN" *Instruction*
 ("OR" *Simple_term* +", " "THEN" *Instruction*) *
 ["ELSE" *Instruction*]
 "END"
 "END"

Assertion_instruction ::=
 "ASSERT" *Predicate* "THEN" *Instruction* "END"

While_instruction ::=
 "WHILE" *Condition* "DO" *Instruction*
 "INVARIANT" *Predicate*
 "VARIANT" *Expression*
 "END"

VIII.4.2.2 Decomposition of non-B0 parameters of existing rules

This predefined behaviour is used with the first rule which has matched during the rule research without considering B0EXPR guards of its WHEN clause. If such a rule has been found, Bart will automatically replace the expressions which have failed at B0EXPR guard checking by local variables in the substitution to refine and will add necessary affectations. This result will be refined again so the rule which has failed previously because of B0EXPR guards will then match successfully.

Bart also checks some constraints on the substitution to refine before saving a rule to be used for this predefined behaviour:

- The substitution has to be an affectation "becomes equal"
- The parts which have to be decomposed must have a simple type (BOOL, INT, a set or an enum)

For example, if we consider the following operation to refine:

example(aa,bb) =
 PRE

```

aa : VALUE &
bb : VALUE &
aa+bb : VALUE
THEN
  rr := bool(aa+bb : sub)
END

```

and the following accessor theory:

```

THEORY_ACCESSOR acs_ensemble IS

  RULE is_in
  REFINES
    @a := bool(@b : @c)
  WHEN
    raffinement_ensemble(@c,@s) &
    B0EXPR(@a) &
    B0EXPR(@b)
  ATTRIBUTES
    out => (@a),
    in => (@b),
    pre => (@b : @s)
  IMPLEMENTATION
    @a := @c_i(@b)
  IMPLEMENT
    @c
  END

END acs_ensemble

```

knowing that sub is an abstract variable refined by *r_ens* (cf. VII.5.1 Figure 26) so the following guard has been added to the hypothesis stack:

```

raffinement_ensemble(sub, VALUE)

```

the only thing that does not match in the rule "is_in" is the guard B0EXPR(@b). Indeed "aa+bb" is not considered as a B0 expression.

So BART will produce the following refinement and the rule can be applied on the second substitution:

```

L_1 := aa + bb ;
rr := bool(L_1 : sub)

```

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 35 : 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 36 : 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  $\wedge$  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  $\vee$  {nn} ||
        TYPE_remaining := TYPE_remaining - {nn} ||
        elt := nn ||
        continue := bool(TYPE_remaining /= {nn})
    END
END
END

```

Figure 37 : 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).

Refinement of iteration machine is done by putting MAGIC pragmas on its variables and operations.

3 elements must be present in machine where iterated set is defined for rules referenced in MAGIC pragma to be applied:

- An abstract constant which type is a permutation on elements of the set
- An operation that returns the cardinal of the set
- An operation that reads the permutation for an element of its domain

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 V CONSTANT_done = CONSTANT[{index1}]
  CONSTANT_remaining ^ CONSTANT_done = {} &
  invariant
VARIANT
  card(CONSTANT_remaining)
END
```

Figure 38 : Invariant iteration generated loop

As for TYPE_ITERATION, $vg_loop := \text{bool}(vg_loop = \text{TRUE} \ \& \ \text{while} = \text{TRUE})$ 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  $\wedge$  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  $\vee$  {nn} ||
    CONSTANT_remaining := CONSTANT_remaining - {nn} ||
    elt := nn ||
    continue := bool(CONSTANT_remaining /= {nn})
  END
END
END

```

Figure 39 : 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 40 : 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 refinement	Associated iteration machine
<i>Machine_i</i>	-
Operation1: No iterator defined	
<i>Machine1_i</i>	<i>Machine1_it</i>
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
<i>Machine2_i</i>	<i>Machine2_it</i>
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 41 : 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" | "newpre") ">=" "(" 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* | *newpre* : If *pre* is used, it contains user defined part of the precondition for the new operation. If *newpre* is used, only given predicate is kept as operation precondition.
- *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 42 : 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	aa <-- add1(bb, cc)	out <-- add1(in1, in2) = PRE

<pre> out => (@a), in => (@b,@c), pre => (@b : INT & @c : INT), body => (@a := @b + @c)) </pre>		<pre> in1 : INT & in2 : INT THEN out <-- in1 + in2 END </pre>
---	--	--

Figure 43 : 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 44 : 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.

Depending on which clause has been used (*pre* or *newpre*), Bart may also automatically add predicates to the operation precondition.

If *newpre* is used, given predicate is instantiated, parameters are replaced (VIII.5.3.2), and the result is simply kept as the new operation precondition.

If *pre* is used, Bart will add new predicates to the precondition.

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* : INTEGER & *cc* : 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 + @c))</pre>	<pre>aa <-- add1(bb, cc)</pre>	<pre>out <-- add1(in1, in2) = PRE in1 : INTEGER & in2 : INTEGER THEN out <-- in1 + in2 END</pre>

Figure 45 : Example of imported operation precondition adding

When using *pre* clause, Bart will also filter content of the generated precondition to keep only predicates that involve a parameter or variable that appears in operation body.

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.5.3.5 Merging generated imported operations

As described in description of Bart parameters, it is possible to activate a mode in which similar generated imported operations will be merged.

The option to use in this purpose is `-O`.

If this mode is active, when an imported operation is about to be generated by Bart, the tool checks if it does not exist yet. If so (existing operation must syntactically identical), previous one will be used.

The new operation is then not added to refinement process, and a call to previous one is inserted in implementation.

Using this mode allows to limit number of generated operations.

On big projects, this number may be very high, and consequently typecheck (particularly checking of promoted operations) may be long.

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 46 : Substitution and rule for LH example

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

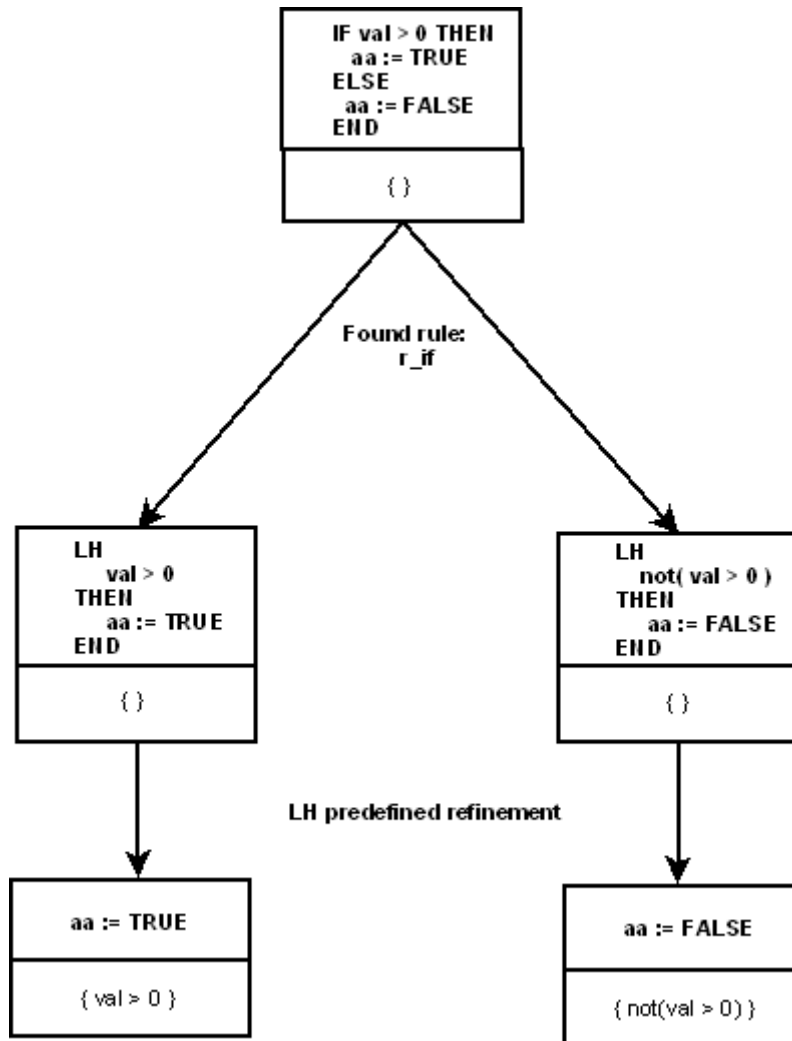


Figure 47 : Example of stack filling with LH substitutions

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 48 : 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 or IMPLEMENTATION clauses of rules. Obviously, declaring new abstract variables in an IMPLEMENTATION clause is not allowed.

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 variables are abstract ones. In this case, REFINED_BY clause may be present to specify which theory or rule must be used. Syntax is "REFINED_BY theory.rule(parameters)" or "REFINED_BY theory". If only a theory name is given, the rule will be searched only within this theory. Parameters usage is identical as for abstract variable refinement. It is possible to define several abstract variables at once so you can specify a unique rule to refine them all. 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. In that case, substitutions forbidden in refinements, as WHILE substitutions and IMPORTED_OPERATION, are naturally prohibited in the result clause.

For example, if we consider the following operation to refine:

```
big_union = BEGIN
  sub1 := sub1 ∨ (sub2 ∨ (sub3 ∨ sub))
END
```

and the following operations rules:

```
THEORY_OPERATION theory_sub IS

  RULE des_union0
  REFINES
    @a := @b ∨ @c
  WHEN
    raffinement_ensemble(@a,@s) &
    raffinement_ensemble(@b,@s)
  REFINEMENT
    @a := @b;
    @a := @a ∨ @c
  END;

  RULE des_union1
  REFINES
    @a := @a ∨ @b
  WHEN
    raffinement_ensemble(@a,@s) &
    bnot(raffinement_ensemble(@b,@s))
  REFINEMENT
    VARIABLE
      #1
    WITH_INV
      #1 <: @s
    WITH_INIT
      #1 := {}
    IN
      #1 := @b;
      @a := @a ∨ #1
  END;

  RULE assign_a_union_b_c_1
  REFINES
    @a := @a ∨ @b
  WHEN
    raffinement_ensemble(@a,@c) &
    raffinement_ensemble(@b,@c)
  IMPLEMENTATION
    TYPE_ITERATION(index => #1,
      type => @c,
      body => (IMPORTED_OPERATION(out => (),
        in => (#1),
```

```

newpre => (#1 : @c),
body => (@a := @a \ / ({#1} /\ (@b)))) ),
invariant => (@a = @a$0 \ / (@c_traites /\ (@b))))
END
END theory_sub

```

knowing that *sub*, *sub1*, *sub2*, *sub3* are abstract variables refined by *r_ens* (cf. VII.5.1 Figure 26) so the following guards have been added to the hypothesis stack:

```

raffinement_ensemble(sub, VALUE)
raffinement_ensemble(sub1, VALUE)
raffinement_ensemble(sub2, VALUE)
raffinement_ensemble(sub3, VALUE)

```

The output generated machines will be:

	Variables	Operations
ex.mch	ABSTRACT_VARIABLES sub1, sub2, sub3, sub INVARIANT sub1 <: VALUE & sub2 <: VALUE & sub3 <: VALUE & sub <: VALUE INITIALISATION sub1 := {} sub2 := {} sub3 := {} sub := {}	big_union = BEGIN sub1 := sub1 \ / (sub2 \ / (sub3 \ / sub)) END
ex_r1.ref	ABSTRACT_VARIABLES abstract_0, sub, sub1, sub2, sub3 INVARIANT abstract_0 <: VALUE INITIALISATION abstract_0 := {} ; sub := {} ; sub1 := {} ; sub2 := {} ; sub3 := {}	big_union = BEGIN /* Rule : theory_sub.des_union1 */ abstract_0 := sub2 \ / (sub3 \ / sub) ; sub1 := sub1 \ / abstract_0 END
ex_r2.ref	ABSTRACT_VARIABLES abstract_0, abstract_1, sub, sub1, sub2, sub3 INVARIANT abstract_1 <: VALUE INITIALISATION abstract_0 := {} ; abstract_1 := {} ; sub := {} ; sub1 := {} ; sub2 := {} ; sub3 := {}	big_union = BEGIN /* Rule : theory_sub.des_union1 */ /* Rule : theory_sub.des_union0 */ abstract_0 := sub2 ; /* Rule : theory_sub.des_union1 */ abstract_1 := sub3 \ / sub ; abstract_0 := abstract_0 \ / abstract_1 ; sub1 := sub1 \ / abstract_0 END
ex_i.imp	[..]	[..]

Figure 49 : Example of abstract variables declared in operation rules

Here, we can see two refinements generated because the rule *des_union1* which declares a new abstract variable is used twice in a row. These newly abstract variables (*abstract_0* and *abstract_1*) are also refined by *r_ens* so the

following guards are added during the refinement process to the hypothesis stack:

```

raffinement_ensemble(abstract_0, VALUE)
raffinement_ensemble(abstract_1, VALUE)

```

The end of the operation refinement is not detailed here.

VIII.7 Usage of substitution rules

VIII.7.1 Accessor rules

VIII.7.1.1 Rule syntax

Accessor rules are similar to operation rules, except that they always implement some variables and import the content of the REFINEMENT/IMPLEMENTATION clause into an imported operation. They are gathered in theories called accessor theories. In order to avoid unintentional refinement errors, the names of every accessor theories from all rule files must be distinct.

The syntax of accessor rules and theories are written below:

```

AccessorTheory
=
  "THEORY_ACCESSOR" ident
  "IS"
  AccessorRule { "," AccessorRule }
  "END" ident
.

AccessorRule
=
  "RULE" ident
  "REFINES" Substitution
  "WHEN" Predicate
  [ "ATTRIBUTES"
    [ "name" ">=" ident "," ]
    [ "out" ">=" "(" [ JokerList ] ")" "," ]
    [ "in" ">=" "(" [ JokerList ] ")" "," ]
    [ ("pre" | "newpre") ">=" "(" Predicate ")" ]
  ]
  ( "REFINEMENT" | "IMPLEMENTATION" ) Substitution
  "IMPLEMENT" JokerList
  "END"
.

```

Syntax 12 : Accessor rule theories

VIII.7.1.2 Role

Accessor rules have originally been created to avoid the splitting deadlocks which can occur when using only classic operation rules. In order to do so, the implementation of the rule is encapsulated into an imported operation. So in refined machine, the substitution is implemented by an operation call, this operation being defined in an imported machine and having as specification and refinement/implementation respectively the REFINES and REFINEMENT/IMPLEMENTATION substitutions instantiated.

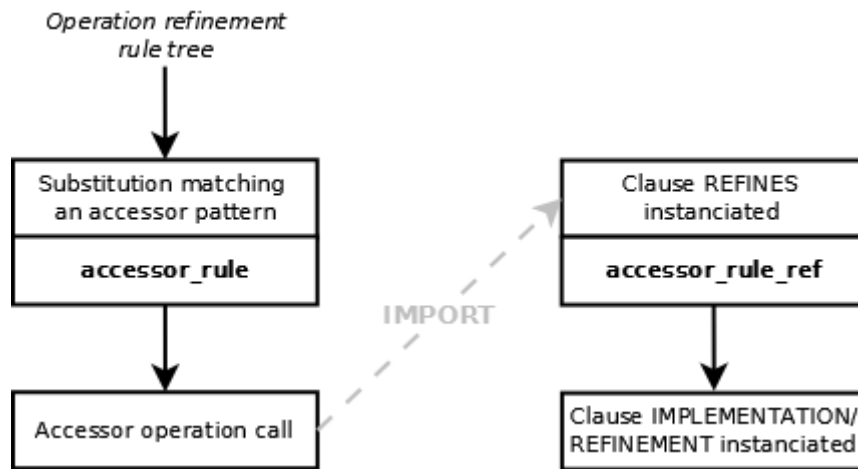


Figure 50 : Refinement rule tree with accessor

To be used properly, an accessor rule must be written every time the refinement of elementary substitution requires to “access” to a refined variable. Moreover, none of the operation rules should implement any refined variables.

During the automatic refinement process, accessor rules are processed just before operation rules. Bart first tries every accessor theories of all rule files and, if no rule can be applied, the research goes on with the operation theories of all rule files.

VIII.7.1.3 Special features

Accessor rules can not contain SUB_REFINEMENT clause. It must have a REFINEMENT or an IMPLEMENTATION and also an IMPLEMENT clause which must define at least one refined variable. The IMPLEMENTATION clause must be used when the variable implemented is only refined by concrete variables. On the opposite, the REFINEMENT clause must be used when the variable implemented

is refined by at least one abstract variable. In the second case, the variable will be refined in an intermediate refinement component in which will appear the refinement result of this rule.

Accessor rules contain a special clause called ATTRIBUTES. It defines the attributes necessary to create the imported operation in which the implementation substitution will be encapsulated. The different parameters of this clause allow the user to define the new operation specification exactly as IMPORTED_OPERATION substitution does.

- *name*: This optional parameter provides a name to the operation. If not present, the following name is automatically computed: `acs_ruleName_firstImplementVariableName`. This name will never be modified, so it has to be unique for each accessor. Indeed, Bart will merge every accessor operations with the same name, even if they come from different rules.
- *out*: (optional) lists output parameters of operation
- *in*: (optional) lists input parameters of operation
- *pre* | *newpre*: (optional) contains the whole predicate that will be operation precondition

As for IMPORTED_OPERATION, Bart automatically adds to the given parameters, the accessed and modified variables which cannot be imported (i.e. local variables, operation parameters, and concrete variables).

In accessor rules, there is no difference between *pre* and *newpre*. In both cases, no automatically computed operation precondition is added to the ones given. If *pre/newpre* parameter is not present, the created imported operation will not have any preconditions.

B0EXPR guards are automatically added to the WHEN conditions for input and output parameters of ATTRIBUTES clause so it is not necessary to write them down.

For example, for the following accessor rule:

```

RULE is_in
REFINES
  @a := bool(@b : @c)
WHEN
  raffinement_ensemble(@c,@s)
ATTRIBUTES
  out => (@a),
  in => (@b),
  pre => (@b : @s)
IMPLEMENTATION
  @a := @c_i(@b)
IMPLEMENT
  @c
END;
```

The real WHEN constraint will be:

```
raffinement_ensemble(@c,@s) &
B0EXPR(@a) &
B0EXPR(@b)
```

VIII.7.1.4 Example 1

If we consider the following operation to refine:

```
switch_element(ee) =
PRE
  ee : VALUE
THEN
  IF ee : sub THEN
    aa := aa - {ee}
  ELSE
    aa := aa \ {ee}
  END
END
```

and the following theories:

```
THEORY_ACCESSOR acs_ensemble IS

  RULE is_in
  REFINES
    @a := bool(@b : @c)
  WHEN
    raffinement_ensemble(@c,@s)
  ATTRIBUTES
    out => (@a),
    in => (@b),
    pre => (@b : @s)
  IMPLEMENTATION
    @a := @c_i(@b)
  IMPLEMENT
    @c
  END;

  RULE add_el_to
  REFINES
    @a := @a \ { @b }
  WHEN
    raffinement_ensemble(@a,@s)
  ATTRIBUTES
    in => (@b),
    pre => (@b : @s)
  IMPLEMENTATION
    @a_i(@b) := TRUE
  IMPLEMENT
    @a
  END;

  RULE rem_el_from
  REFINES
    @a := @a - { @b }
  WHEN
    raffinement_ensemble(@a,@s)
```



```

ATTRIBUTES
  in => (@b),
  pre => (@b : @s)
IMPLEMENTATION
  @a_i(@b) := FALSE
IMPLEMENT
  @a
END

END acs_ensemble
&

THEORY_OPERATION op_if IS

  RULE if_then_else_0
  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;

  RULE if_then_else_1
  REFINES
    IF @a = TRUE THEN
      @b
    ELSE
      @c
    END
  WHEN
    B0EXPR(@a)
  SUB_REFINEMENT
    (LH @a = TRUE THEN @b END) -> (@d),
    (LH not(@a = TRUE) THEN @c END) -> (@e)
  IMPLEMENTATION
    IF @a = TRUE THEN
      @d
    ELSE
      @e
    END
  END

END op_if

```

knowing that sub is an abstract variable refined by r_{ens} (cf. VII.5.1 Figure 26) so the following guard has been added to the hypothesis stack:

```
raffinement_ensemble(sub, VALUE)
```

The refinement will give the following rule tree:

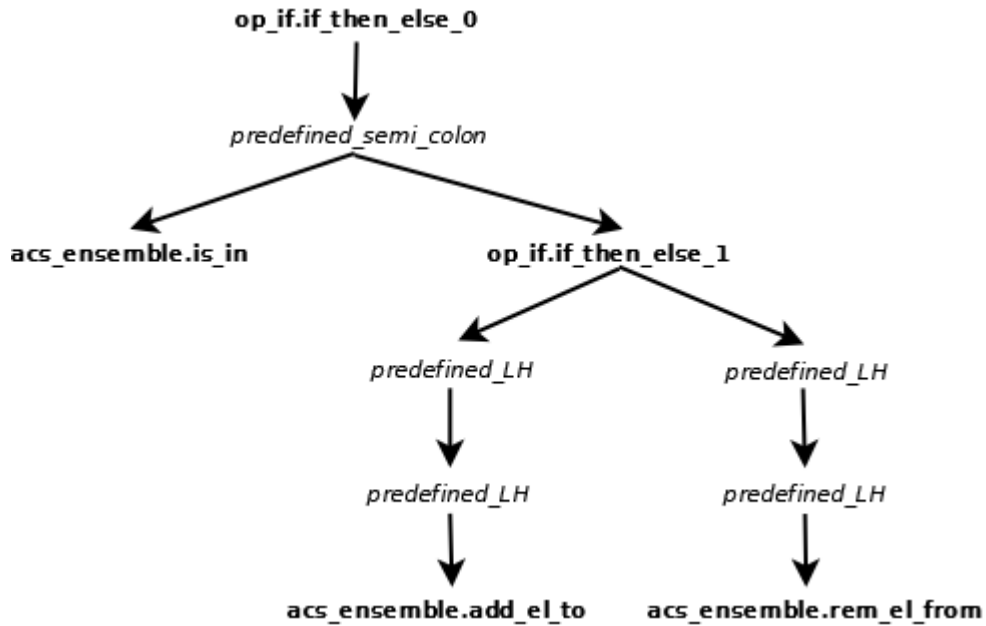


Figure 51 : Rule tree for accessor example

And the output generated machines will be:

Machine <pre> switch_element(ee) = PRE ee : VALUE THEN IF ee : sub THEN sub := sub - {ee} ELSE sub := sub \/ {ee} END END END </pre>	Machine1 <pre> acs_add_el_to_sub(par_in) = PRE par_in : VALUE THEN sub := sub \/ {par_in} END; par_out <-- acs_is_in_sub(par_in) = PRE par_in : VALUE THEN par_out := bool(par_in : sub) END; acs_rem_el_from_sub(par_in) = PRE par_in : VALUE THEN sub := sub - {par_in} END </pre>
Machine_i <pre> switch_element(ee) = VAR l_1 IN l_1 <-- acs_is_in_sub(ee) ; IF l_1 = TRUE THEN acs_rem_el_from_sub(ee) ELSE acs_add_el_to_sub(ee) END END </pre>	Machine1_i <pre> acs_add_el_to_sub(par_in) = BEGIN sub_r (par_in) := TRUE END; par_out <-- acs_is_in_sub(par_in) = BEGIN par_out := sub_r (par_in) END; </pre>

```
END
```

```
acs_rem_el_from_sub(par_in) =
BEGIN
  sub_r (par_in) := FALSE
END
```

Figure 52 : Example of generated accessors

VIII.7.1.5 Example 2

If we consider the following operation to refine:

```
remove_last = PRE
  size(suite) > 0
THEN
  suite := front(suite)
END
```

and the following accessor rules:

```
THEORY_ACCESSOR my_acs_fonc IS

  RULE rem_el_from
  REFINES
    @a := {@p} <<| @a
  WHEN
    fonc(@a,@b,@c,@e)
  ATTRIBUTES
    in => (@p),
    pre => (@p:@b)
  IMPLEMENTATION
    @a_i(@p) := @e
  IMPLEMENT
    @a
  END

END my_acs_fonc
&

THEORY_ACCESSOR my_acs_seq IS

  RULE set_to_front
  REFINES
    @a := front(@a)
  WHEN
    sequence(@a,@s,@m)
  ATTRIBUTES
    pre => (size(@a)>0)
  REFINEMENT
    @a_r := {@a_size_i} <<| @a_r;
    @a_size_i := @a_size_i - 1
  IMPLEMENT
    @a
  END

END my_acs_seq
```

knowing that *suite* is an abstract variable refined by an abstract and a concrete variable according to the rule *r_sequence* (cf. VII.5.2 Figure 28) so the following guard has been added to the hypothesis stack:

```
sequence(suite, VALUE_i, c_max_seq)
```

The output generated machines will be:

	Variables	Operations
seq.mch	VARIABLES suite INVARIANT suite : seq(VALUE_i) & size(suite) <= c_max_seq INITIALISATION suite := []	remove_last = PRE size(suite) > 0 THEN suite := front(suite) END
seq_i.imp		remove_last = BEGIN /* Rule : my_acs_seq.set_to_front */ /*? suite := front(suite) ?*/ acs_set_to_front_suite END
seq_1.mch	ABSTRACT_VARIABLES suite INVARIANT suite : seq(VALUE_i) INITIALISATION suite := []	acs_set_to_front_suite = PRE size(suite) > 0 THEN /* pragma_b MAGIC (my_acs_seq_ref. set_to_front_ref) */ suite := front(suite) END
seq_1_r1.ref	ABSTRACT_VARIABLES suite_r CONCRETE_VARIABLES suite_size_i INVARIANT suite_r : 0..c_max_seq+>VALUE_i & suite_size_i : NAT & suite = 1..suite_size_i< suite_r & suite_size_i = size(suite) INITIALISATION suite_r := {} ; suite_size_i := 0	acs_set_to_front_suite = BEGIN /* Rule : my_acs_seq_ref.set_to_front_ref */ suite_r := {suite_size_i} << suite_r ; suite_size_i := suite_size_i - 1 END
seq_1_i.imp	INITIALISATION suite_size_i := 0	acs_set_to_front_suite = BEGIN /* Rule : my_acs_seq_ref.set_to_front_ref */ /* Rule : my_acs_fonc.rem_el_from */ /*? suite_r := {suite_size_i} << suite_r ?*/ acs_rem_el_from_suite_r(suite_size_i) ; /* Rule : assign_a_minus_b.c.assign_a_minus_b_c_1 */ suite_size_i := suite_size_i - 1 END
[..]	[..]	[..]

Figure 53 : Example of accessor for abstract variable

Here, we can see that the accessor *set_to_front* creates an imported operation which has its specification defined in *seq_1.mch*. In this machine, the variable *suite*, which is “accessed” by the accessor, is still unrefined.

Then, the variables introduced by the refinement of *suite* (*suite_r* and *suite_size_i*) are declared in the refinement *seq_1_r1.ref* and the accessor operation is refined according to the accessor rule.

Finally, the refinement goes on with the implementation of B0 parts ("*suite_size_i* := *suite_size_i* - 1") and another accessor call to modify *suite_r* ("*acs_rem_el_from_suite_r*(*suite_size_i*)") which is not detailed here.

VIII.7.2 Structural and operation rules - Operation refinement

VIII.7.2.1 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 13 : 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.7.2.2 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.

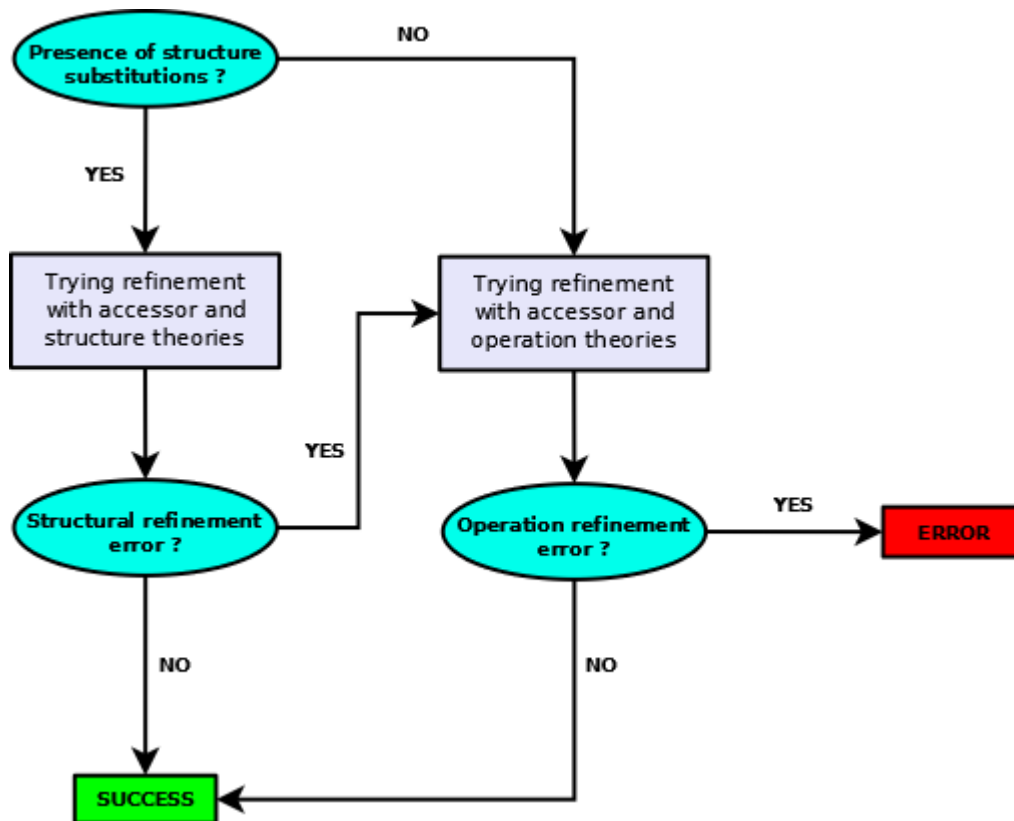


Figure 54 : Usage of structure and operation rules

Bart tries structural refinement if current operation contains structure substitution. A structural refinement error occurs if Bart cannot refine completely the operation with only accessor and structure rules. If such an error occurs, Bart will try to refine the operation with accessor and 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	operation(val) = PRE

<pre> RULE default REFINES @a WHEN 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) match(@b,FALSE)) & bnot(BOEXPR(@a)) IMPLEMENTATION #1 := @b @a := #1 END; END operation </pre>	<pre> val : INTEGER THEN IF val > 0 THEN aa := TRUE ELSE aa := FALSE END END; out <-- affect_true = BEGIN out := TRUE END </pre>
---	---

Figure 55 : Theories and operation for operation refinement example

For these rules and operations, refinement results are:

Operation	Found rules	Produced result
-----------	-------------	-----------------

operation	<ul style="list-style-type: none"> - 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 END</pre>
affect_true	- operation.r_affect_bool_2	<pre>out <-- affect_true = BEGIN out := TRUE END</pre>
operation1 = BEGIN aa := TRUE END	<ul style="list-style-type: none"> - operation.r_affect_bool_1 - operation.r_affect_bool_2 	<pre>operation1 = VAR l_1 IN l_1 = TRUE; aa := l_1 END</pre>
operation2 = BEGIN aa := FALSE END	<ul style="list-style-type: none"> - operation.r_affect_bool_1 - operation.r_affect_bool_2 	<pre>operation2 = VAR l_1 IN l_1 = FALSE; aa := l_1 END</pre>

Figure 56 : 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.7.3 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	=	<pre>"THEORY_INITIALISATION" ident "IS" InitialisationRule { ";" InitialisationRule } "END" ident</pre>
InitialisationRule	=	<pre>"RULE" ident "REFINES" Substitution ["WHEN" Predicate] ("REFINEMENT" "IMPLEMENTATION") Substitution "END"</pre>

Syntax 14 : 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.

The IMPLEMENTATION clause must be used when the variable initialized is only refined by concrete variables. On the opposite, the REFINEMENT clause must be used when the variable initialized is refined by at least one abstract variable. If a variable is refined by both abstract and concrete variables, elements which initialize concrete variables must be put in an IMPLEMENT substitution.

For example, an initialisation rule for a variable of type sequence (cf. VII.5.2 Figure 28) would be:

```

RULE r_init_seq
REFINES
  @a := []
WHEN
  sequence(@a,@s,@m)
REFINEMENT
  @a_r := {};
  IMPLEMENT(@a_size_i := 0)
END

```

Figure 57 : Example of initialisation with REFINEMENT clause

In this example the substitution « @a_r := {} » will be refined as it concerns an abstract variable.

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

- 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 58 : 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	<pre>aa := 1 bb := 0 cc := FALSE</pre>

Figure 59 : 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 15 : 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 60 : 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.

Note: There is no user pass for accessor theories.

IX.2 Tactic theory

IX.2.1 Syntax

```

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

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

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

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

Syntax 16 : 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, operations and accessors). 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 61 : 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 \ / 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, Bart will keep going with remaining tactic patterns.

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.

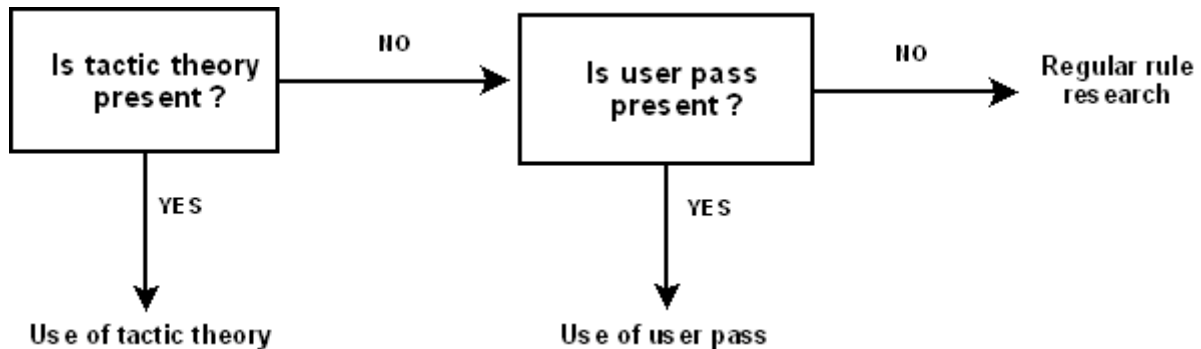


Figure 62 : Usage of tactics and user passes

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</pre>

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

Figure 63 : 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 64 : Refinement result formatting example

X.2 Well-definedness conditions and typing predicates

Two additional options can be used to complete the refinement result. They can be accessed via the command line (-w and -W arguments) or directly via the AtelierB resource file (ATB*BART*WellDefConditions and ATB*BART*TypesInWhile).

The first one (-w / ATB*BART*WellDefConditions) automatically computes the conditions necessary to the well-definedness of the imported operations and

adds them to their preconditions. For now, only the evaluation of functions is treated.

For example, Bart will add the precondition $xx:dom(ff)$ to an operation containing the substitution $aa:=ff(xx)$.

The second one (-W / ATB*BART*TypesInWhile) automatically computes and adds typing predicates in while invariants. This is done only for local variables which are modified in the body of the loop and before the loop.

X.3 Generating refinement components

For each operation to refine, Bart may generate intermediate refinement results which have to appear in refinement components. This can come from different constraints:

- The rule has the clause EXPLICIT_REFINEMENT
- The rule introduces new abstract variables
- The rule implements variables which are refined by at least one abstract variable

As operation refinement is in fact a rule tree, one operation can necessitate several intermediate refinement results to appear in successive refinement components.

However, there is no splitting between the intermediate results of all operations for one level of importation, so there may be conflicts coming from variables refinement.

For example, if an original variable is refined by another abstract variable and is implemented by two different operations but at different refinement steps, it will produce a conflict error and the machines generation will fail. Indeed, all operations have to implement a variable in one same component so the variable can effectively be implemented.

To avoid this kind of conflict, please use accessor rules to implement variables.

You will find below an example showing how the intermediate refinements are computed.

If we consider the following operation to refine:

```
test(ee) = PRE
  ee : VALUE
THEN
  Sub := Sub  $\setminus$  (Sub2  $\setminus$  Sub3)  $\parallel$ 
  boolen := bool(ee : Sub2  $\setminus$  Sub3)
END
```

and the following operation rules:

```

THEORY_OPERATION theory_sub IS

  RULE assign_a_8
  REFINES
    @a := @b
  WHEN
    raffinement_ensemble(@a,@c)
  IMPLEMENTATION
    TYPE_ITERATION(index => #1,
      type => @c,
      body => (IMPORTED_OPERATION(out => (),
        in => (#1),
        newpre => (#1 : @c),
        body => (@a := (@a - {#1}) \/ ((@b) /\ {#1}))) ),
      invariant => (@a = (@a$0 - @c_traites) \/ (@b /\ @c_traites)))
  END;

  RULE des_union0
  REFINES
    @a := @b \/ @c
  WHEN
    raffinement_ensemble(@a,@s) &
    raffinement_ensemble(@b,@s)
  EXPLICIT_REFINEMENT
    @a := @b;
    @a := @a \/ @c
  END;

  RULE des_union1
  REFINES
    @a := @a \/ @b
  WHEN
    raffinement_ensemble(@a,@s) &
    bnot(raffinement_ensemble(@b,@s))
  REFINEMENT
    VARIABLE
      #1
    WITH_INV
      #1 <: @s
    WITH_INIT
      #1 := {}
    IN
      #1 := @b;
      @a := @a \/ #1
  END;

  RULE assign_a_union_b_c_1
  REFINES
    @a := @a \/ @b
  WHEN
    raffinement_ensemble(@a,@c) &
    raffinement_ensemble(@b,@c)
  IMPLEMENTATION
    TYPE_ITERATION(index => #1,
      type => @c,
      body => (IMPORTED_OPERATION(out => (),
        in => (#1),
        newpre => (#1 : @c),
        body => (@a := @a \/ ({#1} /\ (@b)))) ),
      invariant => (@a = @a$0 \/ (@c_traites /\ (@b))))
  END

```

```

END theory_sub
&

THEORY_OPERATION theory_bool IS

  RULE assign_a_bool_and_b_c_1
  REFINES
    @a := bool(@b & @c)
  WHEN
    B0EXPR(@a)
  REFINEMENT
    #1 := bool(@b);
    #2 := bool(@c);
    IMPLEMENT(@a := bool(#1 = TRUE & #2 = TRUE))
  END;

  RULE assign_a_bool_belongs_b_c_23
  REFINES
    @a := bool(@b : @c /\ @d)
  EXPLICIT_REFINEMENT
    @a := bool(@b : @c & @b : @d)
  END;

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

END theory_bool

```

knowing that variables refinement has added the following guards to the hypothesis stack:

```

raffinement_ensemble(Sub, VALUE)
raffinement_ensemble(Sub2, VALUE)
raffinement_ensemble(Sub3, VALUE)
scalar(booleen)

```

The refinement will give the following rule tree:

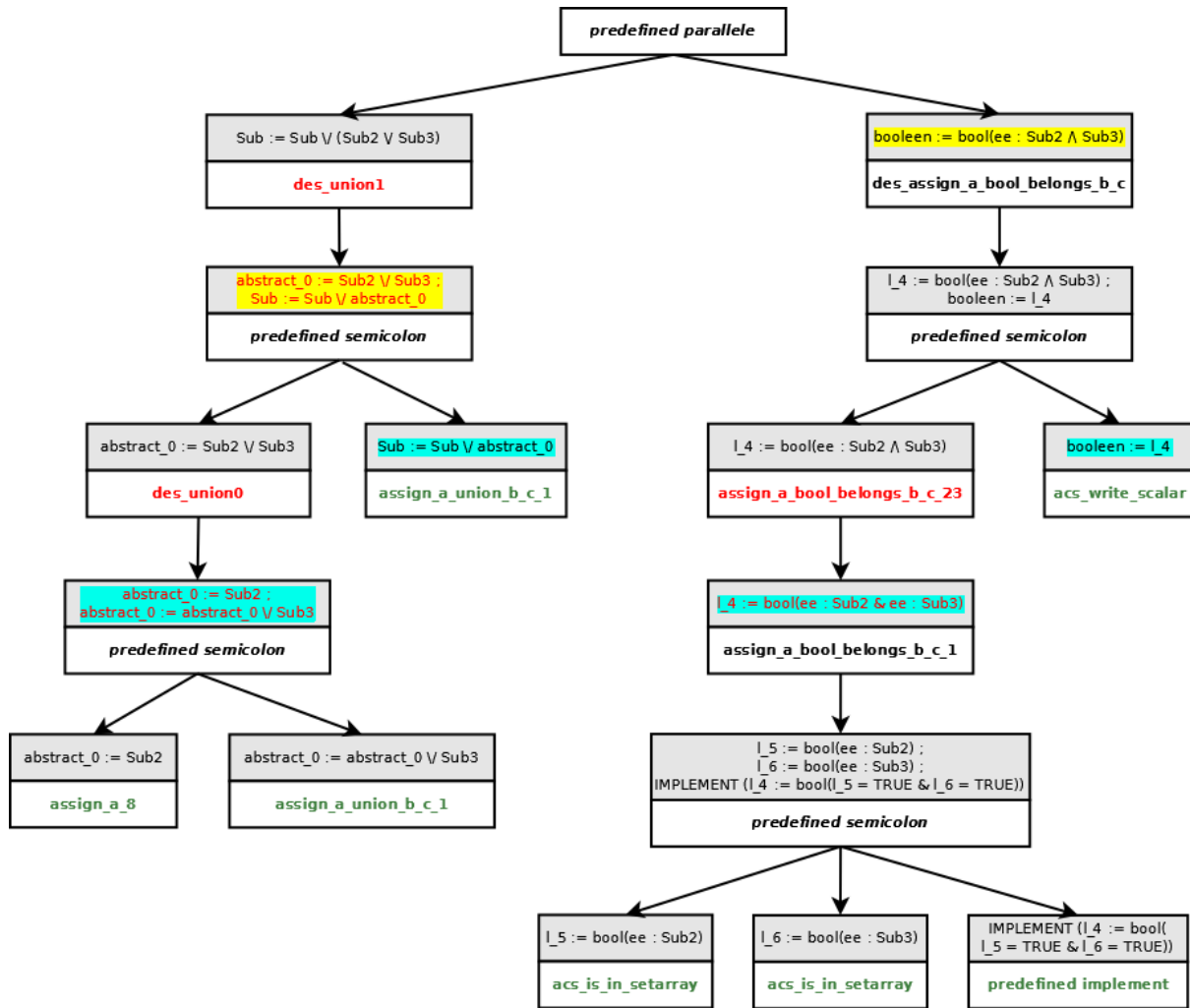


Figure 65 : Rule tree with explicit refinements

You can see in red the rules and their results which have to appear in an intermediate refinement component. There are two rules which have an EXPLICIT_REFINEMENT clause (*des_union0* and *assign_a_bool_belongs_b_c_23*) and one rule which declares a new abstract variable (*des_union1*).

The rules written in green are implementations which may call imported operations but this is not detailed here.

To generate necessary intermediate refinements, Bart processes the rule tree from bottom to top searching rules which have to appear in a refinement component. It regroups the maximum of these rules such as they can be displayed in a same refinement component (two of these rules cannot be in the same branch of the tree). As all branches of the tree have to be represented in each refinement of the operation, Bart automatically adds to the refinement

result the highest substitution of all branches which do not contain any rules that necessitate appearing in a refinement component. So it can generate the two refinement results for this operation which are highlighted in yellow and in blue.

Finally, the output generated machines will be:

	Variables	Operations
ex.mch	ABSTRACT_VARIABLES Sub, Sub2, Sub3, boolean INVARIANT Sub <: VALUE & Sub2 <: VALUE & Sub3 <: VALUE & boolean : BOOL INITIALISATION Sub := {} Sub2 := {} Sub3 := {} boolean :: BOOL	test(ee) = PRE ee : VALUE THEN Sub := Sub \vee (Sub2 \vee Sub3) boolean := bool(ee : Sub2 \wedge Sub3) END
ex_r1.ref	ABSTRACT_VARIABLES Sub, Sub2, Sub3, abstract_0, boolean INVARIANT abstract_0 <: VALUE INITIALISATION Sub := {} ; Sub2 := {} ; Sub3 := {} ; abstract_0 := {} ; boolean :: BOOL	test(ee) = BEGIN abstract_0 := Sub2 \vee Sub3 ; Sub := Sub \vee abstract_0 ; boolean := bool(ee : Sub2 \wedge Sub3) END
ex_r2.ref	ABSTRACT_VARIABLES Sub, Sub2, Sub3, abstract_0, boolean INITIALISATION Sub := {} ; Sub2 := {} ; Sub3 := {} ; abstract_0 := {} ; boolean :: BOOL	test(ee) = VAR l_4 IN abstract_0 := Sub2 ; abstract_0 := abstract_0 \vee Sub3 ; Sub := Sub \vee abstract_0 ; l_4 := bool(ee : Sub2 & ee : Sub3) ; boolean := l_4 END
ex_i.imp	[..]	[..]

Figure 66 : Example of generated refinement components

X.4 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.4.1 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, $\text{exported}(op)$ are variables exported by operation op , and $\text{implement}(op)$ 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 $\text{exported}(op)$ and $\text{implement}(opX)$ is not empty

Accessor operations are pushed to the end of the output chain in a dedicated component.

X.4.2 Resolving deadlocks

X.4.2.1 Bart splitting algorithm

This section presents the algorithm used by Bart to split operations with respect for constraints exposed in X.4.1.

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 $\text{implement}(op)$ 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:

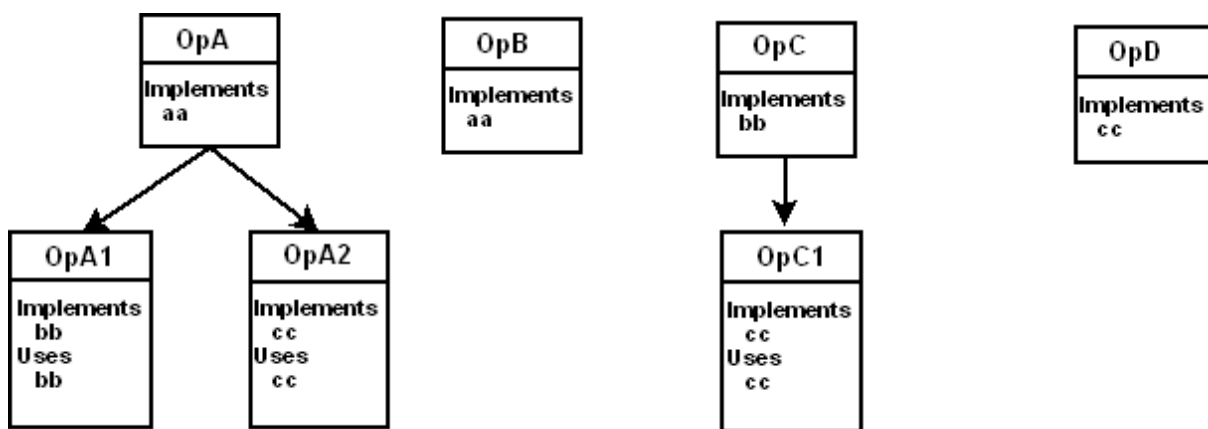


Figure 67 : Operations to implement for splitting example

For these operations, Bart may generate following machines:

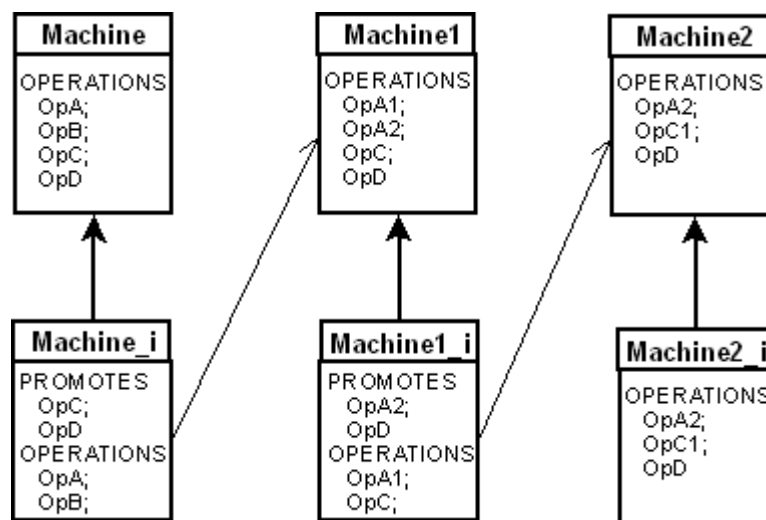


Figure 68 : Result machines for splitting example

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 69 : Splitting process example

X.4.2.2 What is a splitting deadlock?

A splitting deadlock is an error in the process previously described in X.4.2.1.

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:

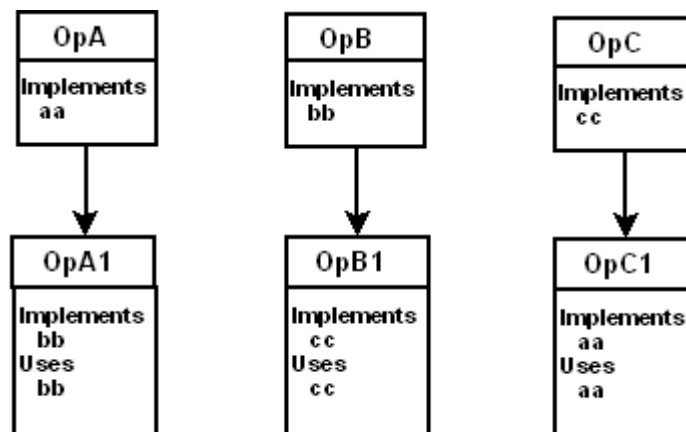


Figure 70 : Splitting deadlock example

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.4.2.3 Solving a deadlock case

Bart is not able to solve automatically the problem. 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.4.2.2. 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.....	8
Figure 2 : Example of Bart standard output	9
Figure 3 : Example of Bart verbose output mode.....	9
Figure 4 : Example of selection of operations to refine	11
Figure 5 : Refinement process order	12
Figure 6 : Examples of pattern-matching without previous instantiation	14
Figure 7 : Examples of @a + @b pattern-matching with previous instantiation	14
Figure 8 : Hypothesis stack for constraint checking examples	15
Figure 9 : Examples of constraint checking	15
Figure 10 : Bart testing rule process	16
Figure 11 : Hypothesis stack filling with environment.....	18
Figure 12 : Example of Bart output components.....	18
Figure 13 : Bart expression guards.....	21
Figure 14 : Bart predicate guards	22
Figure 15 : Bart substitution guards	22
Figure 16 : Bart heterogeneous guards	23
Figure 17 : Predicate theory example.....	24
Figure 18 : Substitution for EMPILE_PRE and DEPILE_PRE example	25
Figure 19 : Example of stack evolution with EMPILE_PRE and DEPILE_PRE	26
Figure 20 : Processing variable theories to find rules	31
Figure 21 : Searching variables refined by a particular rule.....	31
Figure 22 : Theories and stack for variable rule research example	32
Figure 23 : Variable rule research example.....	33
Figure 24 : Rules for type predicate example	33
Figure 25 : Example of type predicates adding	33
Figure 26 : Example of variable refinement rule with variable implementation .	35
Figure 27 : Variable refinement rule with concrete variable.....	36
Figure 28 : Variable refinement rule with abstract variable	36
Figure 29 : Theories and stack for operation rule research.....	40
Figure 30 : Example of operation rule research	40
Figure 31 : Process of substitution refinement	42
Figure 32 : Substitution and theories for rule tree example	44
Figure 33 : Example of refinement rule tree.....	44
Figure 34 : Bart predefined refinement behaviours	46
Figure 35 : Type iteration generated loop, without while parameter.....	52
Figure 36 : Type iteration generated loop, with while parameter	52
Figure 37 : Type iteration generated machine	53
Figure 38 : Invariant iteration generated loop	54
Figure 39 : Invariant iteration generated machine	55

Figure 40 : Concrete iteration generated loop	56
Figure 41 : Example of generated iterators	57
Figure 42 : Imported operation naming example	59
Figure 43 : Simple imported operation example	59
Figure 44 : Imported operation example with parameters adding	60
Figure 45 : Example of imported operation precondition adding	61
Figure 46 : Substitution and rule for LH example	63
Figure 47 : Example of stack filling with LH substitutions	64
Figure 48 : Local variable declaration example	65
Figure 49 : Example of abstract variables declared in operation rules	67
Figure 50 : Refinement rule tree with accessor	69
Figure 51 : Rule tree for accessor example	73
Figure 52 : Example of generated accessors	74
Figure 53 : Example of accessor for abstract variable	75
Figure 54 : Usage of structure and operation rules	77
Figure 55 : Theories and operation for operation refinement example	78
Figure 56 : Operation refinement example	79
Figure 57 : Example of initialisation with REFINEMENT clause	80
Figure 58 : Substitution and theories for initialisation refinement example	81
Figure 59 : Initialisation refinement example	81
Figure 60 : User pass theory example	82
Figure 61 : Tactic theory example	84
Figure 62 : Usage of tactics and user passes	85
Figure 63 : Bart refinement result formatting	87
Figure 64 : Refinement result formatting example	87
Figure 65 : Rule tree with explicit refinements	91
Figure 66 : Example of generated refinement components	92
Figure 67 : Operations to implement for splitting example	94
Figure 68 : Result machines for splitting example	94
Figure 69 : Splitting process example	95
Figure 70 : Splitting deadlock example	95

XII APPENDIX B – SYNTAX ELEMENTS TABLE

Syntax 1 : Predicate theory	24
Syntax 2 : Rule files	28
Syntax 3 : Variable rule theories	30
Syntax 4 : Operation rule theories	39
Syntax 5 : Type iteration	51
Syntax 6 : Invariant iteration syntax	54
Syntax 7 : Concrete iteration	56
Syntax 8 : Seen operation	57
Syntax 9 : Imported operation	58
Syntax 10 : Implement	62
Syntax 11 : LH	62
Syntax 12 : Accessor rule theories	68
Syntax 13 : Structure theories	76
Syntax 14 : Initialisation theories	79
Syntax 15 : User pass theory	82
Syntax 16 : Tactic theory	83

XIII APPENDIX C - CONFIGURING BART IN ATELIERB WITH RESOURCES

ATB*BART*RefinerFile: Path to PatchRefiner to use.

ATB*BART*OptimizationMode: True or false, indicates if imported operations must be merged when identical

ATB*BART*HeaderFile: Predefined header file

ATB*BART*WellDefConditions: True or false, indicates if well-definedness conditions must be added to imported operation preconditions

ATB*BART*TypesInWhile: True or false, indicates if typing predicates must be added to while invariants

XIV APPENDIX D – RULE FILES COMPLETE SYNTAX

This section presents the complete Bart rule files syntax.

XIV.1 Rule files

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

Theory

=

```
VariableTheory
| OperationTheory
| AccessorTheory
| StructureTheory
| InitialisationTheory
| UserPassTheory
| TacticTheory
| PredicateTheory
```

.

XIV.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" JokerList
  [ "REFINED_BY" ident [ "." ident "(" Expression ")" ] ]
  "WITH_INV" Predicate
  "END"
.

```

XIV.3 Initialisation refinement rules

```

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

InitialisationRule
=
  "RULE" ident
  "REFINES" Substitution
  [ "WHEN" Predicate ]
  ( "REFINEMENT" | "IMPLEMENTATION" ) Substitution
  "END"
.

```

XIV.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" | "EXPLICIT_REFINEMENT" | "IMPLEMENTATION" )

```

```

        { RefinementVarDecl }
        Substitution
["IMPLEMENT" IdentOrJokerList ]
"END"
.

RefinementVarDecl =
(
    ("VARIABLE" | "ABSTRACT_VARIABLE") VarDeclList
    [ "REFINED_BY" ident [ "." ident "(" Expression ")" ] ]
|
    "CONCRETE_VARIABLE" vardecl )
    "WITH_INV" Predicate
    "WITH_INIT" Substitution
    "IN"
.

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

```

XIV.5 Accessor refinement rules

```

AccessorTheory
=
    "THEORY_ACCESSOR" ident
    "IS"
    AccessorRule { ";" AccessorRule }
    "END" ident
.

AccessorRule
=
    "RULE" ident
    "REFINES" Substitution
    "WHEN" Predicate
    [ "ATTRIBUTES"
        [ "name" "=>" ident "," ]
        [ "out" "=>" "(" [ JokerList ] ")" "," ]
        [ "in" "=>" "(" [ JokerList ] ")" "," ]
        [ ("pre" | "newpre") "=>" "(" Predicate ")" ]
    ]
    ( "REFINEMENT" | "IMPLEMENTATION" ) Substitution
    "IMPLEMENT" JokerList
    "END"
.

```

XIV.6 Structural refinement rules

```

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

```


XIV.7 User pass theory

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

XIV.8 Tactic theory

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

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

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

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

XIV.9 Predicate synonyms theory

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

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

XIV.10 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" | "newpre") => "(" 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>
  ")"

```

•

—

—

"_"/_"/././.

BinaryExpression20 = BinaryExpression115 (" " BinaryExpression

BinaryExpression115 =

BinaryExpress

Version: 1.6

```

|      "{" 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"
.

```

XIV.13 Diverse

```

JokerList = joker { ",", joker }.

IdentOrJokerList = IdentOrJoker { ",", IdentOrJoker }.

IdentOrJoker = ident | joker.

IdentJokerVardeclList =
    IdentOrJokerOrVardecl { ",", IdentOrJokerOrVardecl }
.

IdentOrJokerOrVardecl = ident | joker | vardecl.

IdentList = ident { ",", ident }.

```