# b2llvm: B developments onto the LLVM

David Déharbe[2]

Forall — Formal Methods and Research Lab.
UFRN — Federal University of Rio Grande do Norte
INES — INC&T para Engenharia de Software, Brazil

September 29th, 2014

[2]j.w.w. J. Souza Neto, R. Bonichon, E. Cid, T. Lecomte, A. Martins Moreira, V. Medeiros

# Overview

# Outline

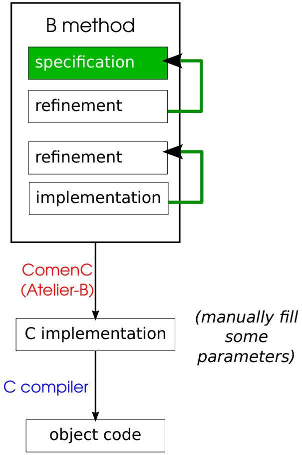
# Introduction

- Design of software components ;
- Software engineering for safety-critical systems :
  - Mass transit sub-systems,
  - Nuclear energy,
  - Aeronautics;
- B method ;
- LLVM compilation framework ;
- Cooperation between academia (UFRN) and industry (Clearsy).

# Introduction

- Design of software components SBCARS;
- Software engineering for safety-critical systems SBES:
  - Mass transit sub-systems,
  - Nuclear energy,
  - Aeronautics;
- B method SBMF;
- LLVM compilation framework SBLP;
- Cooperation between academia (UFRN) and industry (Clearsy).

# Motivation, 1




- *Problem:* certify source code synthesis in Atelier-B
- *Approach:* redundancy, by developing a new code generator to compare with.

# Motivation, 2

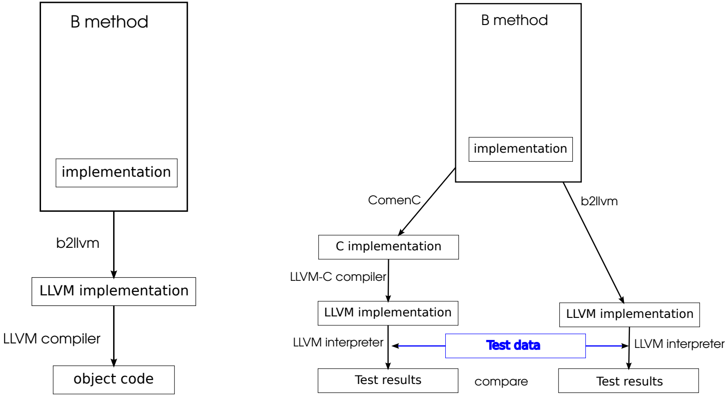
```
IMPLEMENTATION timer_i
REFINES timer_r
IMPORTS
    hh.counter, mm.counter, ss.counter
...
```

Errors (21 Warnings (0)

| Message | Location |
|---|---|
| timer_i | |
| Unsupported renaming of abstract machine | Line 6, Column 5 |
| Unsupported renaming of abstract machine | Line 6, Column 17 |
| Unsupported renaming of abstract machine | Line 6, Column 29 |

- *Problem:* arbitrary limitations in current code synthesis
- *Approach:* remove limitations, keeping in mind application context.

# The b2llvm project




- Code synthesis for B targeting LLVM.
- Working prototype.

# Outline

# B: a stand-alone module

## Specification

```
MACHINE counter
VARIABLES value, overflow
INVARIANT
    value: INT & overflow: BOOL &
    value : 0.. MAXINT & ((overflow = TRUE) => (value = MAXINT))
INITIALISATION
    value := 0 || overflow := FALSE
OPERATIONS
    zero = value, overflow := 0, FALSE
    inc =
    CHOICE
      value :: 0..(MAXINT-1) || overflow := FALSE
    OR
      value := MAXINT || overflow := TRUE
    END;
    res <-- get = res := value
END
```

# B: a stand-alone module

## Implementation

```
IMPLEMENTATION counter_i
REFINES counter
CONCRETE_VARIABLES value, error
INVARIANT value: INT & error : BOOL & (error = overflow)
INITIALISATION  value := 0; error := FALSE
OPERATIONS
    zero = BEGIN
        value := 0;
        error := FALSE
    END;
    inc = IF value < MAXINT THEN value := value + 1
          ELSE error := TRUE END;
    res <-- get = res := value
END
```

# B: a compound module

## Specification

```
MACHINE wd
CONSTANTS timeout
PROPERTIES timeout : INT & timeout > 0
VARIABLES ticker
INVARIANT ticker : INT & ticker >= 0 & timeout >= ticker
INITIALISATION ticker := 0
OPERATIONS
  start = ticker := timeout;
  tick = IF ticker > 0 THEN
      ticker := ticker - 1
    END;
  res <-- expired = IF ticker = 0 THEN
      res := TRUE
    ELSE
      res := FALSE
    END
END
```

# B: a compound module

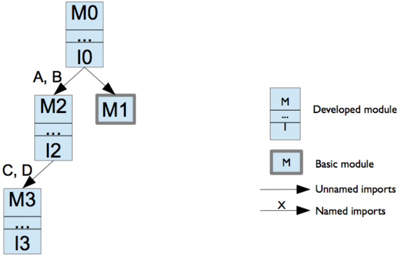
## Implementation

```
IMPLEMENTATION wd_i
REFINES wd
VALUES timeout=50
IMPORTS counter
INVARIANT overflow = FALSE & timeout - value = ticker
INITIALISATION
   VAR count IN
       count := 0;
       WHILE count < timeout DO
           inc; count := count+1
       INVARIANT value = count
       VARIANT timeout - count
       END
   END
OPERATIONS
   tick =
   VAR elapsed, diff IN
       elapsed <-- get;
       diff := timeout - elapsed;
       IF diff > 0 THEN inc END
END;
```

# Some remarks on the B method

- Construction of software *modules*.
- A module may be
  - a *developed* module: implementation derived in B.
  - a *base* module: implementation derived outside of B.
- A developed module has a B specification (machine) and a B implementation.
- A base module has a B specification (machine).
- A B module may *import* other modules.
- A root B module may (transitively) contain several instances of other B modules.

# B projects




1. Modules access other modules (base or developed) through their interface.
2. For a given top-level module, the number of sub-module instances is fixed statically.

# Outline

# LLVM

*"The LLVM Project is a collection of modular and reusable compiler and toolchain technologies."* (*http://www.llvm.org*)

- Used in many commercial and open-source projects
  - Adobe, Apple, Cray, Intel, NVIDIA
- Used in academic research
  - *Formalizing the LLVM Intermediate Representation for Verified Program Transformations*. J. Zhao, S. Nagarakatte, M. Martin, S. Zdancewic. POPL 2012.
- backbone of LLVM: an *assembly language*
  - an intermediate representation (IR)
  - generated by compiler front-ends (parsing, type analysis)
  - consumed, transformed by back-ends (optimization, target code generation).

# LLVM-IR

- Example

C

```c
void inc(int * pi)
{
 *pi += 1;
}
```

LLVM

```llvm
define void @inc(i32* %pi) {
entry:
  %0 = load i32* %pi
  %1 = add i32 %0, 1
  store i32 %1, i32* %pi
  ret void
}
```

- Single-static assignment
- Strongly typed
- Modular

# LLVM-IR syntax: excerpts

$$
\begin{array}{rcl}
\textit{module} & ::= & \textit{item}^{+} \\
\textit{item} & ::= & \textit{type\_def} \mid \textit{const\_decl} \mid \textit{const\_def} \mid \textit{var\_def} \\
 & & \mid \textit{function\_decl} \mid \textit{function\_def} \\
\textit{type\_def} & ::= & \textit{name}\ \texttt{=}\ \texttt{type}\ \textit{type} \\
\textit{type} & ::= & \texttt{void} \mid \textit{itype} \mid \{\ \textit{type}^{+}\ \} \mid \textit{type}* \mid \texttt{opaque} \\
\textit{const\_decl} & ::= & \textit{name}\ \texttt{=}\ \texttt{external constant}\ \textit{type} \\
\textit{const\_def} & ::= & \textit{name}\ \texttt{=}\ \texttt{constant}\ \textit{type}\ \textit{iliteral} \\
\textit{var\_def} & ::= & \textit{name}\ \texttt{=}\ \texttt{common global}\ \textit{type}\ \texttt{zeroinitializ} \\
\textit{function\_decl} & ::= & \texttt{declare}\ \textit{type}\ \textit{name}\ (\ \textit{type}^{+}\ ) \\
\textit{function\_def} & ::= & \texttt{define}\ \textit{type}\ \textit{name}\ (\ \textit{param}^{+}\ )\ \{\ \textit{block}^{+}\ \} \\
\textit{param} & ::= & \textit{type}\ \textit{name} \\
\textit{block} & ::= & \textit{lbl}\ :\ \textit{inst}^{+}
\end{array}
$$

# LLVM-IR syntax: some instructions

$$
\begin{array}{rcl}
\textit{inst} & ::= & \textit{name} \texttt{ = alloca } \textit{type} \\
& | & \textit{name} \texttt{ = } \textit{arith itype exp} \texttt{ , } \textit{exp} \\
& | & \textit{name} \texttt{ = icmp } \textit{rel} \texttt{ i1 } \textit{exp} \texttt{ , } \textit{exp} \\
& | & \textit{name} \texttt{ = call } \textit{type} \texttt{ ( } \textit{arg}^{+} \texttt{ )} \\
& | & \textit{name} \texttt{ = getelementptr } \textit{type} * \textit{exp}\texttt{, } \textit{index}\texttt{, } \textit{index} \\
& | & \textit{name} \texttt{ = load } \textit{type exp} \\
& | & \texttt{store } \textit{type exp}\texttt{, } \textit{type} * \textit{exp} \\
& | & \texttt{br i1 } \textit{exp} \texttt{ , label } \textit{lbl} \texttt{ , label } \textit{lbl} \\
& | & \texttt{br label } \textit{lbl} \\
& | & \texttt{switch } \textit{type exp} \texttt{ , branch } \textit{lbl} \texttt{ [ } \textit{branch}^{+} \texttt{ ]} \\
& | & \texttt{ret } \textit{type exp} \\
& | & \texttt{ret void}
\end{array}
$$

# LLVM-IR expressions: expressions, misc.

$$
\begin{array}{rcl}
\mathit{arith} & ::= & \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{sdiv} \mid \texttt{srem} \\
\mathit{rel} & ::= & \texttt{eq} \mid \texttt{ne} \mid \texttt{sgt} \mid \texttt{sge} \mid \texttt{slt} \mid \texttt{sle} \\
\mathit{exp} & ::= & \mathit{name} \mid \mathit{literal} \mid \\
 & & \texttt{getelementptr} \ ( \ \mathit{type} \ \mathit{exp} \ , \ \mathit{index} \ , \ \mathit{index} \ ) \\
\mathit{index} & ::= & \mathit{itype} \ \mathit{iliteral} \\
\mathit{branch} & ::= & \mathit{iliteral} \ \mathit{iliteral} \ \mathit{lbl} \\
\mathit{arg} & ::= & \mathit{type} \ \mathit{exp}
\end{array}
$$

# LLVM module items

- type declaration

  ```
  @bignum = type opaque
  ```

- Type definitions

  ```
  @bool = type i1
  @int = type i32
  @rational = type {@bool, @int, @int}
  @Prational = type @rational*
  ```

- constant declaration

  ```
  @maxint = external constant i32
  ```

- constant definition

  ```
  @zero = constant i32 0
  ```

- function declaration

  ```
  declare i32 @putchar(i32)
  ```

- function definition...

# LLVM function definitions

```
define void @f1(@Prational %P) {
entry:
  %g = alloca i32                                    ;; int g
  %1 = getelementptr @Prational %P, i32 0, i32 2
  %2 = load i32* %1
  %3 = icmp ne i32 %2, 0
  br i1 %3, label %label0, label %label1             ;; if (P->d != 0) {
label0:
  %4 = getelementptr @Prational %P, i32 0, i32 1
  call void @gcd(i32* %g, i32 %4, i32%2)             ;; gcd(&g, P->n,
  %5 = load i32* %g                                  ;;     P->d);
  %6 = load i32* %4
  %7 = sdiv i32 %4, %5
  store i32 %7, i32* %4                              ;; P->n /= g;
  %8 = load i32* %2
  %9 = sdiv i32 %8, %5
  store i32 %9, i32* %1                              ;; P->d /= g;
  br label1                                          ;; }
label1:
  ret void                                           ;; return;
}
```

# LLVM tools

| | |
|---|---|
| lli | LLVM bitcode interpreter |
| llvm-as | assembles LLVM-IR to LLVM bitcode |
| llvm-link | links LLVM bitcode files |
| opt | optimizer |
| llc | static compiler (architecture dependent) |
| lldb | debugger |
| | ... profiling, coverage, archiver ... |

# Outline

# Some general requirements

- No dynamic memory allocation
  - This is a requirement in some applications (often safety-critical systems).
  - The size of the data space of a B component is statically bounded.
  - No need for a third-party library.
- Separate code generation
  - This is a requirement for large projects.
  - Internal changes in a B module shall only require generating code for that module.
  - Changes in the interface of a B module requires generating code for that module and those other modules that use it.

# Code generation

- Specified as a set of code generation rules[3]

$$\|\mathsf{a}\|^{Beq} \equiv \mathbf{let}\ l \otimes p \otimes t' = \|\mathsf{i.lhs}\|^{lv}\ \mathbf{and}\ r \otimes v \otimes t \mathrel{\hat{=}} \|\mathsf{i.rhs}\|^{Expr}\ \mathbf{in}$$

$$\texttt{"}\boxed{l}$$

$$\boxed{r}$$

$$\texttt{store}\ \boxed{t}\ \boxed{v}\texttt{,}\ \boxed{t'}\ \boxed{p}\ \downarrow\texttt{"}$$

- Validated via
  - trial encoding
  - rapid prototyping
    - read XML files produced by a third-party tool
    - Python
  - *ad hoc* testing.

[3]You will be spared a detailed presentation of the rules.

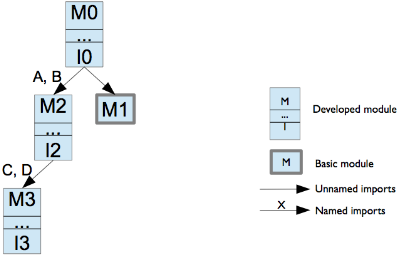
# B constructs

- Constants, variables
- Module instantiation (imports)
- Imperative constructs: assignment, conditional, loop, local variables, operation call.
- Arithmetic and relational expressions, conditionals.
- Integers, Booleans, enumerations.
- $+$ Arrays
- - Lambda expressions, records,
- - - Machine parameters.

# B constructs

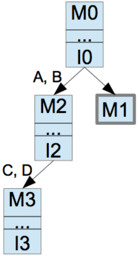
- Constants, variables
- Module instantiation (imports) $\leftarrow$ More on this
- Imperative constructs: assignment, conditional, loop, local variables, operation call.
- Arithmetic and relational expressions, conditionals.
- Integers, Booleans, enumerations.
- $+$ Arrays
- - Lambda expressions, records,
- - - Machine parameters.

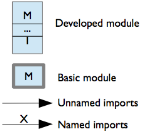
# B projects and code generation




1. modules access other modules (base or developed) through their interface
   - generate LLVM for module *interface*.
2. For a given top-level module, the number of sub-module instances is fixed statically.
   - global variables.
   - two code generation modes: *component* and *project*.

# B projects and code generation




To build the project LLVM code:

b2llvm component mode:

- M0 M2 M3 $\Longrightarrow$ `m0.ll m2.ll m3.ll`

b2llvm project mode:

- M0 $\Longrightarrow$ `init-m0.ll`

LLVM optimize, generate assembly, link

- `init-m0.ll m0.ll m2.ll m3.ll m1.o` $\Longrightarrow$ `a.out`

# Bricks of LLVM code

**type def:** A structure type (one item per state variable and import): `%M$state$ = type {` *type*$^+$ `}`

**type decl:** `%M$state$ = type opaque`

**type ref:** A pointer type: `%M$ref$ = type %M$state$*`

**function decl:** An initialization function:
`declare void @M$init$(%M$ref$,` *type*$^+$`)`
One function for each operation in the module:
`declare void @M$op(%M$ref$,` *type*$^+$`)`

**function def:** (for developed modules):

```
define void @M$op(%M$ref$ %self$, param+) {
    block+
    exit:  ret void
}
```

# Bricks of LLVM code

**type def:** A structure type (one item per state variable and import): `%M$state$ = type {` *type*$^+$ `}`

**type decl:** `%M$state$ = type opaque`

**type ref:** A pointer type: `%M$ref$ = type %M$state$*`

**function decl:** An initialization function:
`declare void @M$init$(%M$ref$,` *type*$^+$`)` ← address of module instance
One function for each operation in the module:
`declare void @M$op(%M$ref$,` *type*$^+$`)`

**function def:** (for developed modules):

```
define void @M$op(%M$ref$ %self$, param+) {
    block+
    exit:  ret void
}
```

# Example 1

```
IMPLEMENTATION counter_i
REFINES counter
CONCRETE_VARIABLES value, error
INVARIANT value: INT & error : BOOL & (error = overflow)
INITIALISATION  value := 0; error := FALSE
OPERATIONS
    zero = BEGIN
        value := 0;
        error := FALSE
    END;
    inc = IF value < MAXINT THEN value := value + 1
          ELSE error := TRUE END;
    res <-- get = res := value
END
```

- type definition

```
%counter$state$ = type { i32, i1 }
%counter$ref$ = type %counter$state$*
```

- function declarations:

```
declare void @counter$init$(%counter$ref$)
declare void @counter$inc(%counter$ref$)
declare void @counter$get(%counter$ref$, i32*)
```

# Example 1 (a function definition)

```
inc = IF value < MAXINT THEN value := value + 1
      ELSE error := TRUE END;
```

```
define void @counter$inc(%counter$ref$ %self$) {
entry:
  %0 = getelementptr %counter$ref$ %self$, i32 0, i32 0
  %1 = load i32* %0
  %2 = icmp slt i32 %1, 2147483647
  br i1 %2, label %label0, label %label1
label0:
  %3 = getelementptr %counter$ref$ %self$, i32 0, i32 0
  %4 = load i32* %3
  %5 = add i32 %4, 1
  %6 = getelementptr %counter$ref$ %self$, i32 0, i32 0
  store i32 %5, i32* %6
  br label %exit
label1:
  %7 = getelementptr %counter$ref$ %self$, i32 0, i32 1
  store i1 1, i1* %7
  br label %exit
exit:
  ret void
}
```

# Example 1 (another function definition)

```
res <-- get = res := value
```

```
define void @counter$get(%counter$ref$ %self$, i32* %res) {
entry:
  %0 = getelementptr %counter$ref$ %self$, i32 0, i32 0
  %1 = load i32* %0
  store i32 %1, i32* %res
  br label %exit
exit:
  ret void
}
```

# Example 2

```
IMPLEMENTATION wd_i
REFINES wd
VALUES timeout=50
IMPORTS counter
INVARIANT overflow = FALSE & timeout - value = ticker
INITIALISATION
   VAR count IN
       count := 0;
       WHILE count < timeout DO
           inc; count := count+1
       INVARIANT value = count
       VARIANT timeout - count
       END
   END
OPERATIONS
   tick =
   VAR elapsed, diff IN
       elapsed <-- get;
       diff := timeout - elapsed;
       IF diff > 0 THEN inc END
END;
```

- type definition

  `%wd$state$ = type { %counter$ref$ }`

- function declarations:

```
declare void @wd$init$(%wd$ref$, %counter$ref$)
declare void @wd$tick(%wd$ref$)
```

# Example 2 (definition of function for initialization)

```
define void @wd$init$(%wd$ref$ %self$, %counter$ref$ %arg0$) {
entry:
  %count = alloca i32                                ;; local variable
  %0 = getelementptr %wd$ref$ %self$, i32 0, i32 0
  store %counter$ref$ %arg0$, %counter$ref$* %0
  call void @counter$init$(%counter$ref$ %arg0$)     ;; initializes counter
  store i32 0, i32* %count                           ;; count := 0
  br label %label1
label1:                                              ;; WHILE
  %1 = load i32* %count
  %2 = icmp slt i32 %1, 50                           ;;    VALUE < 50
  br i1 %2, label %label2, label %label0
label2:                                              ;; DO
  %3 = getelementptr %wd$ref$ %self$, i32 0, i32 0
  %4 = load %counter$ref$* %3
  call void @counter$inc(%counter$ref$ %4)           ;;    inc
  %5 = load i32* %count
  %6 = add i32 %5, 1
  store i32 %6, i32* %count                          ;;    count := count+1
  br label %label1                                   ;; END
label0:
  br label %exit
exit:
  ret void
}
```

# Putting it all together: component mode

- type definition
  - need type references of imported modules
- function definitions
  - need type references of transitively imported modules (initialization)
  - need function declarations of imported modules (operation calls, initialization)

# Putting it all together: project mode

- to create instances of all the instances of all the modules:
  - type definition of module
  - type reference of module
  - type definitions of transitively imported modules
  - type references of transitively imported modules

  For each instance Q of a module M

  ```
  @$Q[path] = common global @$Q$state$
  zeroinitializer
  ```
- to initialize the system:
  - function declaration (initialization) of the module

  ```
  define void @$init$(void) {
      call void @M$init$($M, ...)
      ret void
  }
  ```

# Putting it all together: project for example 2

```
%counter$state$ = type {i32, i1}
%counter$ref$ = type %counter$state$*
%wd$state$ = type {%counter$ref$}
%wd$ref$ = type %wd$state$*
@$wd = common global %wd$state$ zeroinitializer
@$counter = common global %counter$state$ zeroinitializer
declare void @wd$init$(%wd$ref$, %counter$ref$)
declare void @wd$tick(%wd$ref$)
define void @$init$() {
entry:
  call void @wd$init$(%wd$ref$ @$wd, %counter$ref$ @$counter)
  ret void
}
```

# Outline

# Possible approaches

- Formal verification:
  - Formalize LLVM-IR (available)
  - Formalize B (partly available)
  - Formalize and verify code generation rules (to be done)
- Assertions in generated code.
  - Implement assert in LLVM.
  - Define code generation rules for full conditions (not only implementable conditions).
- Test generated code (ongoing work)
- Generate debugging aids (some done)
- Traceability (done)

# Possible approaches

- Formal verification:
  - Formalize LLVM-IR (available)
  - Formalize B (partly available)
  - Formalize and verify code generation rules (to be done)
- Assertions in generated code.
  - Implement assert in LLVM.
  - Define code generation rules for full conditions (not only implementable conditions).
- Test generated code (ongoing work) ←
- Generate debugging aids (some done)
- Traceability (done) ←

# Model based testing and b2llvm

- BETA (model-based testing for B)
  - generates test guides in C
  - several coverage criteria
- B-method
  - design a correct B implementation
- b2llvm
  - generates a LLVM implementation of the module
  - emits C declarations for the module interface
- Test guides are applied to the generated LLVM implementation.

# Traceability: an example

- Human review
- Automatically insertion of comments
- Comments include reference to source B implementation

```
;;1 The type for the state of "counter" is defined in "counter_i",
;; it is an aggregate such that:
;;1.1 Position "0" represents variable "value".
;;1.2 Position "1" represents variable "error".
%counter$state$ = type {i32, i1}
;;2 The type for references to state encodings of "counter" is:
%counter$ref$ = type %counter$state$*
;;3 The function implementing initialisation for "counter" is
;;  named "@counter£init£" and has the following parameters:
;;3.1 "%self£": address of LLVM aggregate storing state of "counter";
define void @counter$init$(%counter$ref$ %self$) {
```

# Outline

# Conclusion

- b2llvm is open-source (`http://www.b2llvm.org/b2llvm`).
- b2llvm is compatible with Atelier-B 4.2 .
- Available validation strategies:
  - output code may be annotated.
  - tests may be automatically generated (experimental).
  - debugging aids may be generated.

# Goals

- Extend B implementation constructs beyond what is currently available in commercial tools;
- Integration with test-generation tool;
- Prove the correctness of the code generation rules

? QUESTIONS ?

COMMENTS ? DOUBTS

? SUGGESTIONS ?