The ATL Virtual Machine

An introduction to the ATL Virtual Machine
V1.0 draft

Frédéric Jouault, and Freddy Allilaire

ATLAS group (INRIA & LINA), University of Nantes, France
http://www.sciences.univ-nantes.fr/lina/atl/

Outline

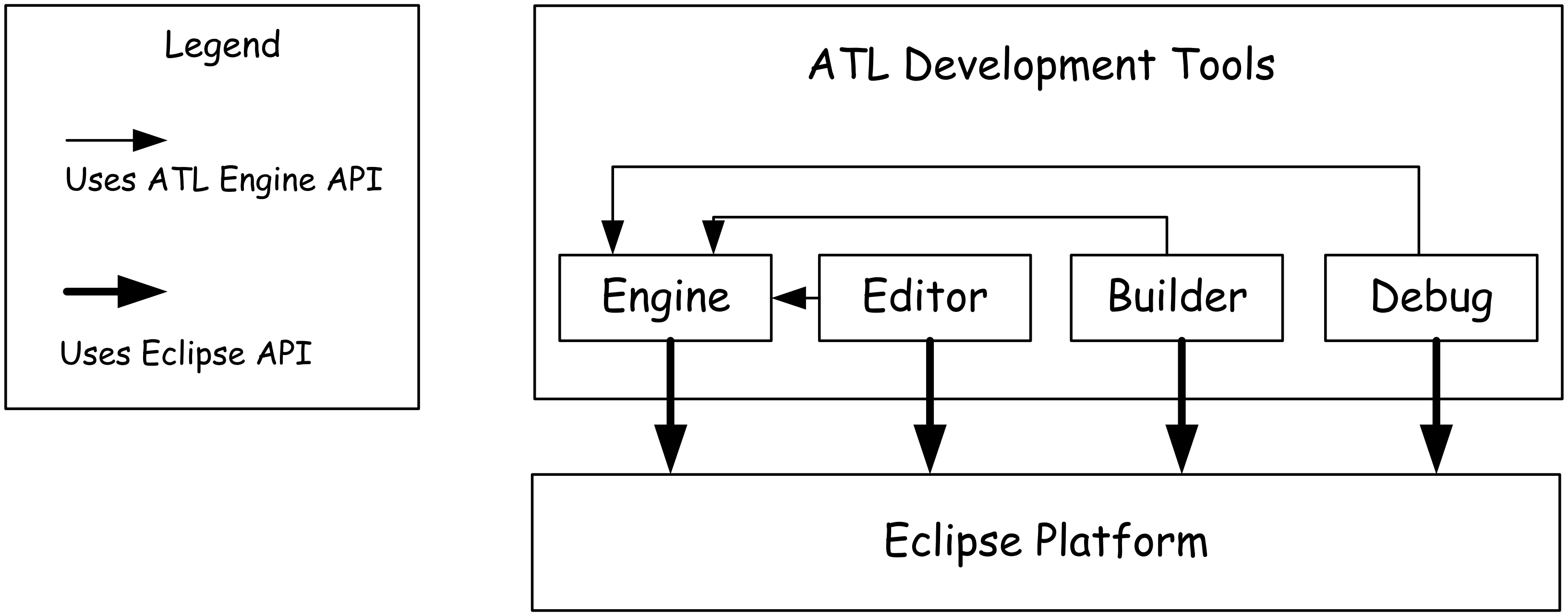
- Introduction
- Structure of the ATL Virtual Machine
- Instruction Set Summary
- ASM File Format
- Compiling for the ATL VM → ACG

Outline

- Introduction
- Structure of the ATL Virtual Machine
- Instruction Set Summary
- ASM File Format
- Compiling for the ATL VM ➔ ACG

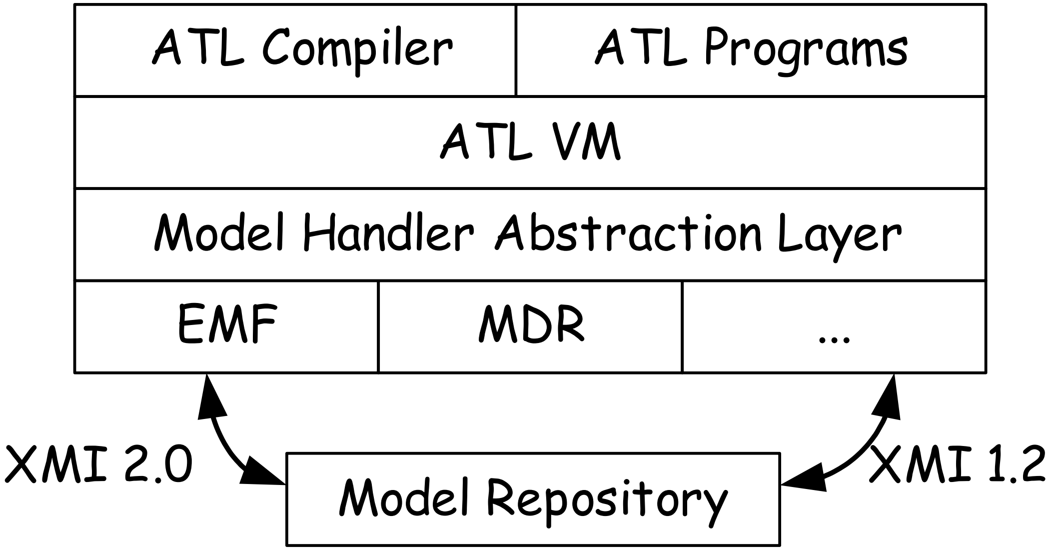
ATL Development Tools (ADT)

- ATL is accompanied by a set of tools built on top of the Eclipse platform
- ADT is composed of the ATL transformation engine (*Engine* block) and the ATL IDE



ATL Engine

- The ATL engine is responsible for dealing with core ATL tasks:
  - Transformation compilation
  - Transformation execution
- The ATL compiler and compiled transformations run on top of the ATL Virtual Machine



ATL VM Introduction – General Steps

- ATL transformations are compiled to programs in specialized byte-codes stored in the ASM file format
- This byte-code is executed by the ATL Virtual Machine (VM)
- The VM is specialized in handling models and provides a set of instructions for model manipulation.
- The VM may run on top of various model management systems (e.g., Eclipse EMF, and Netbeans MDR)
  - To isolate the VM from their specifics, an intermediate level is introduced called *Model Handler Abstraction Layer*
  - This layer translates the instructions of the VM for model manipulation into instructions of a specific model handler

ATL VM Introduction – Technical Steps

- The ATL VM is an abstract computing machine
  - It is similar to the Java VM
  - Its instructions operate on an operand stack
  - It has its own model-oriented instruction set
- The ATL VM does not depend on ATL
  - Other languages can be implemented on top of it
- Instructions are stored in the ASM file format
  - This format is based on XML

Outline

- Introduction
- Structure of the ATL VM
- Instruction Set Summary
- ASM File Format
- Compiling for the ATL VM → ACG

Structure of the ATL Virtual Machine

- The following aspects are going to be detailed:
  - Data Types
  - Runtime Data Structures
  - Stack Frames
  - Representation of Model Elements

Datatype

- There are two kinds of datatypes:
  - Primitive (Boolean, Integer, Real, String)
  - Composite (Tuple, Collections, Map, metamodel elements, etc.)
- The VM performs runtime type checking
- A compiler may perform compile-time type checking to catch problems earlier
- The instruction set distinguishes its operand types using instructions intended to operate on values of specific types

Datatype: Primitive Types

- Primitive types defines the core types the VM relies on
  - Boolean: encodes *true* and *false* Boolean values
  - Integer: encodes integral numerical values
  - Real: encodes real numerical values
  - String: encodes string values
- These types support the operations defined in the OCL standard library

Datatype: Composite Types

- Tuple
  - Represents OCL tuples
- Collections
  - Specify collections of elements
  - Exist in several versions: Bag, OrderedSet, Sequence, Set
- Map
  - Represents associative tables
- Metamodel elements
  - Correspond to the types defined in the metamodels of the source and target models

Runtime Data Structures

- ATL VM defines various runtime data areas used during execution
- PC register
  - Program counter
  - Contains the address of the ATL VM instruction currently being executed (except when executing *native* (typically Java) operations)
- Call Stack
  - Created at start-up
  - Stores stack frames
  - Analogous to the call stack of conventional languages
  - Holds variables and partial results (in an operand stack)
  - Plays a part in operation invocation and return

Stack Frames

- Created each time a non-native operation is invoked
- Destroyed when its operation invocation completes
- Stored on the call stack
- Each frame has its own array of local variables and its own operand stack
- Only one frame is active at anytime and is referred to as the current frame
- A frame ceases to be the current frame if its operation completes or invokes another operation

Stack Frames: Local variables

- Each frame is associated with a set of variables
  - ➔ Local variables
- Can be hold values of primitive or composite type
- Used to pass arguments to instructions
- Used to pass arguments to invoked operations
  - The values of the arguments associated with an operation invocation are subsequently assigned to local variables of the new stack frame created for the invoked operation

Stack Frames: Operand stack

- Each frame contains a LIFO stack
  - ➔ the operand stack
- The instruction set contains instructions to load constants or values from variables or fields onto the operand stack
- Other instructions take operands from the operand stacks, operate on them, and push the result back on the operand stack
- It is also used to prepare arguments to be passed to operations and to receive operation results
- Each entry on the operand stack can hold a value of any ATL VM type
- Values from the operand stack must be operated upon in ways appropriate to their types

Representation of Model Elements

- The way model elements are internally managed by ATL virtual machine is not constrained
- Model handler implementers are therefore free to use some existing model handling facilities or to provide their own model repository
- Custom implementations may be based on various underlying technologies
- The current version provides model handlers for Eclipse/EMF and Netbeans/MDR

Outline

- Introduction
- Structure of the ATL VM
- Instruction Set Summary
- ASM File Format
- Compiling for the ATL VM → ACG

Instruction Set Summary

- An ATL VM instruction consists of an *opcode* specifying the operation to be performed followed an optional inline *operand*
- Additional arguments or data that may be required by an instruction have to be fetched from the top of the operand stack
- The instructions of the ATL virtual machine can be grouped into three distinct sets:
  - stack handling
  - control
  - model handling

Operand Stack Handling Instructions

- The ATL virtual machine provides a number of instructions enabling direct manipulations of the operand Stack
- They may be sorted into three subgroups:
  - Pushing a constant onto the operand stack: *push*, *pushi*, *pushd*, *pusht*, *pushf*.
  - Untyped manipulations of the operand stack: *pop*, *dup*, *dup_x1*, *swap*.
  - Untyped loading & storing a variable to/from the operand stack: *load*, *store*

Control Instructions

- Control instructions cause the ongoing execution to continue from an instruction that may not be the instruction that follows the current instruction
- The ATL virtual machine defines 4 different control instructions:
  - Conditional branch: *if*
    - Takes a Boolean value from the operand stack
  - Unconditional branch: *goto*
  - Iterative execution: *iterate, enditerate*
    - Operates on Collections
  - Method invocation: *call*

Model Handling Instructions

- These instructions are dedicated to models and model elements handling
- This instruction set also enables the ATL virtual machine to handle other composite types like Tuples
- There are 4 model handling instructions:
  - Create a new element: *new*
  - Access element properties: *get, put*
  - Find a metamodel element: *findme*
  - Access the ATL context module element: *getasm*
    - e.g., used to called helpers defined in the context of the module

Outline

- Introduction
- Structure of the ATL VM
- Instruction Set Summary
- ASM File Format
- Compiling for the ATL VM → ACG

ASM File Format

- This format is the one which is interpreted by the current reference implementation of the ATL Virtual Machine
- An ASM file can contain the compiled version of an ATL transformation, an ATL query, or an ATL library
- An ASM file can also contain the compiled version of another language
- The ASM file format is an XML-based textual format

ASM Overview

- The *asm* element is an ordered structure that contains the transformation constant pool followed by a set of field and one or more operation
- The *asm* element also has a *name* attribute
  - The value of this attribute is a constant pool index pointing to the constant pool entry that stores the transformation name
- The *asm* element contains a non empty set of operations
- The operation set specifies the instructions to be executed by the ATL virtual machine in order to carry out the compiled transformation
  - Execution starts at the *main* operation

The constant pool

- It stores all the constant values, whatever their type,
- Each constant value can be addressed by an index in the constant pool
- The constant pool is composed of *constant* elements
- These *constant* elements have a *value* attribute that contains the constant value
- This value is encoded as a string

Data Types Encoding

- The ASM file format defines an internal encoding for the types of the elements it handles
- These values, encoded as strings, are used to specify data types for the *type* attribute elements *field*, *context*, and *parameter*
  - The *field* element defines an attribute of the ATL context module
  - The *context* element specifies the element type for which an operation is defined
  - The *parameter* element defines an operation parameter

Operation Signature

- It is used by the *call* instructions to identify the operation to be invoked
- It has to encode all information that may be required by the virtual machine to match an operation call to its corresponding operation definition
- In the ASM file format, the signature type is encoded by means of a string and relies on the same type encoding used to represent types

Expressions Location Encoding

- Within each defined operation, the *linenumbertable* element contains the bindings between source code expressions and ATL Virtual Machine instructions
- Each *linenumbertable* entry (element *lne*) binds a set of consecutive virtual machine instructions to a portion of the source code
- The ASM file format defines a specific string encoding to identify source code portions:
  - <start-line>:<start-column>-<end-line>:<end-column>
- This encoding defines both the portion start line and start column and its end line and end column

The Fields

- An *asm* element can include some *field* elements
- Fields can be viewed as global variables that are directly associated with the ATL context module
- Fields have a *name* and a *type* attributes
- These attributes contain valid constant pool indexes
- Value of the *name* attribute indexes a field name
- The *type* attribute points to a data type encoding constant

The Operations

- An ASM file contains a set of operation
- Executable ASM files (e.g., ATL queries, or ATL transformations, but not ATL libraries) have a program entry point which must be implemented by a *main* operation
- Each operation contains a sequence of instructions that will have to be interpreted by the ATL Virtual Machine when the operation will be invoked

The *operation* element

- Is an ordered structure in which are specified a context, some parameters, some instructions (within the *code* element), a line number table and a local variable table
- The *context* element defines the context in which the operation is defined
- The set of parameters encodes both type and name of the operation parameters
- The *code* element contains an ordered sequence of instructions that implements the treatment associated with the operation
- The *linenumbertable* element defines bindings between the instructions of the code element and the expressions that appear in the source code
- Finally, the *localvariabletable* stores names of the local variables that are defined for the operation

The *context* element

- An operation context element specifies the context in which the operation is defined
- Each operation is associated with a unique context
- This context is defined by the *type* attribute that points to a data type encoding entry of the constant pool

The *parameters* element

- Each operation is associated with an ordered list of *parameter* elements inside the *parameters* element
- The *parameters* element is empty for operations that accept no parameter
- Each *parameter* element has a *name* and a *type* attributes
- Both attributes contain the index of a constant pool entry
- The *name* attribute points to a constant pool entry that contains a variable name
- The *type* attribute refers to a data type encoding constant, which defines the parameter type

The *code* element

- The treatments that are performed by an operation are defined within the operation *code* element
- The code part of an operation definition contains a sequence of instructions among those that have been defined in the ATL virtual machine instruction set
- Each defined instruction has its counterpart (in the form of an instruction element) in the ASM file format
  - The optional *arg* attribute contains the inline parameter of the instruction, if any
  - Its value is an index that targets a constant pool entry
  - Depending on the instruction, the *arg* attribute points to different kinds of constant pool entries:
    - A constant of appropriate type for push instructions
    - *An integer value for load* and *store* instructions, which identifies an index in the local variable table
    - *An integer value representing a brancihng offset for the if* and *goto* instructions
    - *A field name for the get* and *set* instructions
    - *An operation signature for the call* instructions

The Line Number Table

- The *linenumbertable* element defines the bindings between the instructions of an operation and the corresponding code within the source file
- This element encodes useful information for implementers that wish to provide debugging facilities along with the ATL Virtual Machine
- The line number table is composed of line number entries (*lne* elements)
- Each entry specifies a binding between an instructions sequence from the operation stack and a part of the source code file
- Each entry has the following attributes: *id*, *begin,* and *end*
- The *id* attribute contains a constant pool entry index that identifies a source file portion
- The *begin* and *end* attributes make it possible to identify the sequence of instructions that are associated with the identified portion of the source code file:
  - They respectively specify the address of the first and of the last instructions of this sequence within the *code* element
  - The instructions are numbered from 0 to the number of instructions minus one
- The *begin* and *end* attributes do not refer to constant pool entries, but directly encode the instruction number values

The Local Variable Table

- The structure is similar to the Java local variable table
- It encodes the name and scope of each local variable within the operation
- A local variable table typically contain at least one entry, which corresponds to the contextual element (i.e., self in OCL, or this in Java) on which the operation has been called.
- Each entry (*lve* element) is associated with a single local variable defined for the current operation
- The *begin* and *end* attributes respectively specify the address of the first and the last instructions for which the variable exists (directly, not as constant pool indexes)
- The *slot* attribute contains the slot number (directly)
- The *name* attribute contains a constant pool entry index specifying the name of the variable
- The first slot of the local variable table (i.e., slot 0) corresponds to the contextual element of the operation
- Subsequent slots are associated with the operation parameters in the order they have been defined

Outline

- Introduction
- Structure of the ATL VM
- Instruction Set Summary
- ASM File Format
- Compiling for the ATL VM → ACG

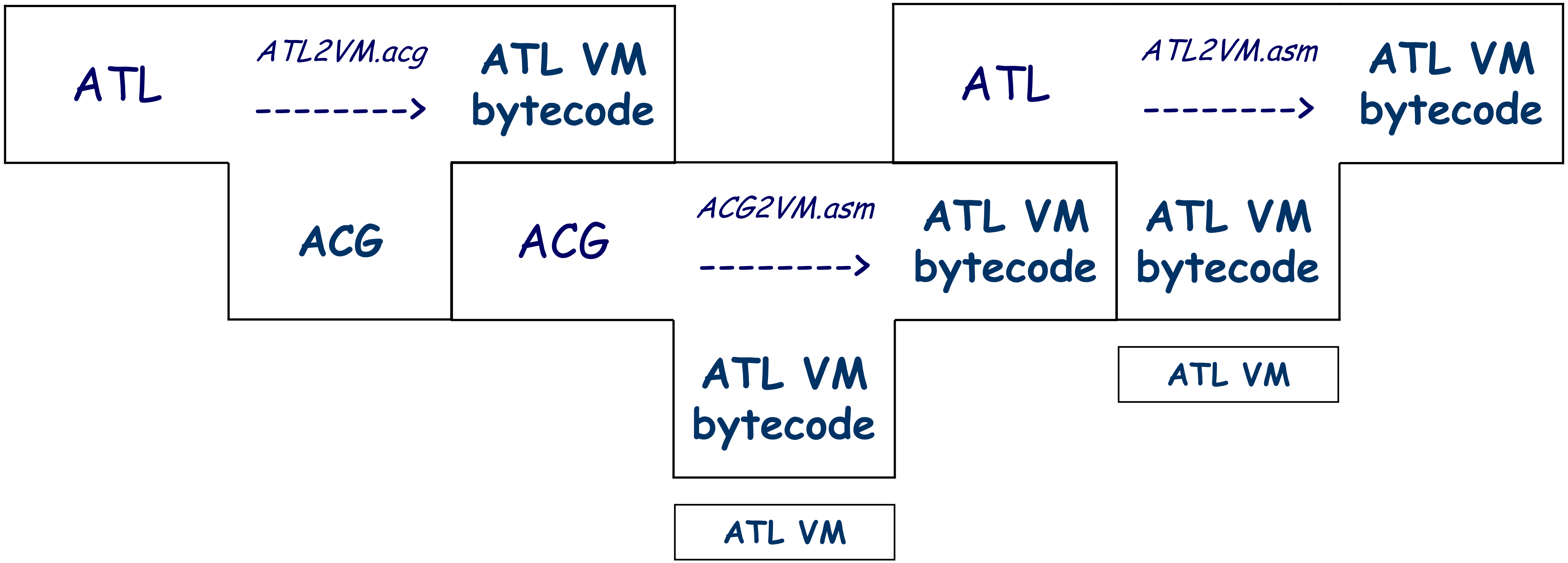
Compiling for the ATL VM → ACG

- ATL Code Generation (ACG) language
  - A transformation DSL with fixed target: ATL Virtual Machine
  - Directly supports generation of ATL VM bytecodes
  - Automatically fills *line number* and *local variable* tables (i.e., debug information)
  - Source model (i.e., program to compile) navigation is faster compared to old compiler and uses a simplification of OCL
  - Example:

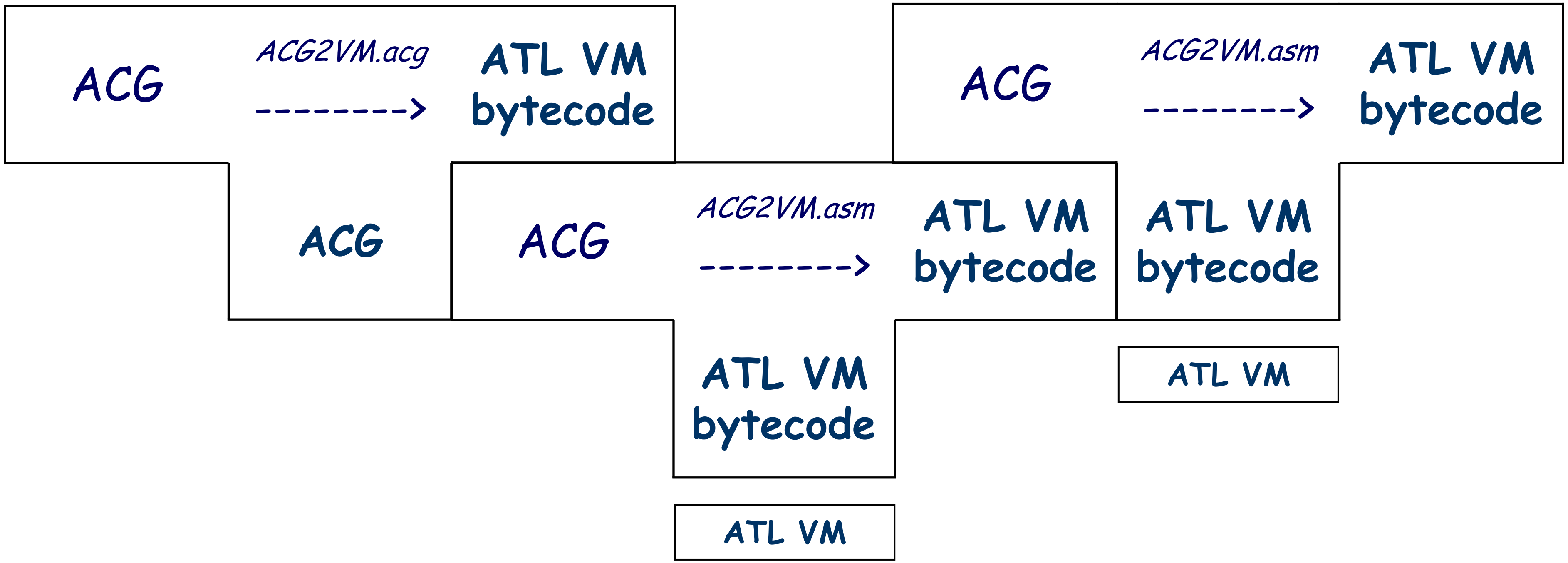
```
code IntegerExp {
  pushi self.integerSymbol
}
```

- *ACG2VM.acg* is the ACG compiler: ACG is bootstrapped
- *ATL2VM.acg* is the new ATL compiler (for ATL 2006)

ATL Compiler Written in ACG



Bootstrapping ACG Compiler



The first version of ACG2VM.asm was obtained by interpreting ACG2VM.acg with an ATL program.

Why ACG and not directly bootstrapping ATL

- ATL is:
  - A Domain Specific Language (DSL) for the domain of model transformation,
  - But in this domain, it is a General Purpose Transformation Language,
  - ➔ An ATL compiler in ATL is possible.
- ACG is:
  - A DSL for the domain of ATL VM bytecode generation,
  - Specifically tuned for this purpose,
  - ➔ An ATL compiler in ACG is simple.
- Additional benefits:
  - ACG can be used to compile any model transformation DSL,
  - Such a DSL then runs on top of ATL VM:
    - It works on every model handler for which there is an ATL VM (and/or driver),
    - It can be source-level debugged like ATL, with the same tools,
    - Etc.

ATL2VM.acg excerpts

```
-- Primitive Literal
code IntegerExp {
    pushi self.integerSymbol
}
-- Collection Literal
code SequenceExp {
    push 'Sequence'
    push '#native'
    new
    analyze self.elements {
          call 'CJ.including(J):CJ'
    }
}
-- Conditional
code IfExp {
    analyze self.condition
    if thn
          analyze self.elseExpression
          goto eoi
     thn:
          analyze self.thenExpression
     eoi:
}
```

```
-- Variables
code LetExp {
    analyze self.variable.initExpression
    variable self.variable named
       self.variable.varName {
          analyze self.in_
    }
}
code VariableExp {
    load self.referredVariable
}
-- Iterator
code IteratorExp |
         self.name = 'exists' and
         self.iterators.size() = 1 {
    pushf
    analyze self.source
    iterate
         variable self.iterators.first() named
             self.iterators.first().varName {
                  analyze self.body
                  call 'B.or(B):B'
         }
    enditerate
}
```

End of the lesson

- Thanks
  - Questions?
  - Comments?

ATLAS group, INRIA & LINA, Nantes