1. ATL Transformation Example: Make → Ant

The Make to Ant example describes a transformation from a Makefile to a file in Ant.

1.1. Transformation overview

The aim of this transformation is to generate a file for the build tool Ant starting from a Makefile.

```
#test of a makefile

CC=gcc
CFLAGS=-Wall –ansi
LDFLAGS=-Wall –ansi

hello : hello.o main.o
	$(CC) -o hello hello.o main.o $(LDFLAGS)
	@skip

hello.o : hello.c
	$(CC) -o hello.o -c hello.c $(CFLAGS)

main.o : main.c hello.h
	@$(CC) -o main.o -c main.c $(CFLAGS)

clean :
	rm -rf *.o

mrproper : clean
	rm -rf $(EXEC)
```

Figure 1. Example of Makefile

```
<project name="Hello">
  <description>Test d'un fichier makefile</description>
  <property name="CC" value="gcc"/>
  <property name="CFLAGS" value="-Wall -ansi"/>
  <property name="LDFLAGS" value="-Wall -ansi"/>
  <target name="hello" depends="hello.o,main.o">
    <echo message="$(CC) -o hello hello.o main.o $(LDFLAGS)"/>
    <exec executable="$(CC) -o hello hello.o main.o $(LDFLAGS)"/>
    <exec executable="skip"/>
  </target>
  <target name="hello.o" >
    <echo message="$(CC) -o hello.o -c hello.c $(CFLAGS)"/>
    <exec executable="$(CC) -o hello.o -c hello.c $(CFLAGS)"/>
  </target>
  <target name="main.o" >
    <exec executable="$(CC) -o main.o -c main.c $(CFLAGS)"/>
  </target>
  <target name="clean" >
    <echo message="rm -rf *.o"/>
    <exec executable="rm -rf *.o"/>
  </target>
  <target name="mrproper" depends="clean">
```

```
    <echo message="rm -rf $(EXEC)"/>
    <exec executable="rm -rf $(EXEC)"/>
  </target>

</project>
```

Figure 2. The corresponding file in Ant

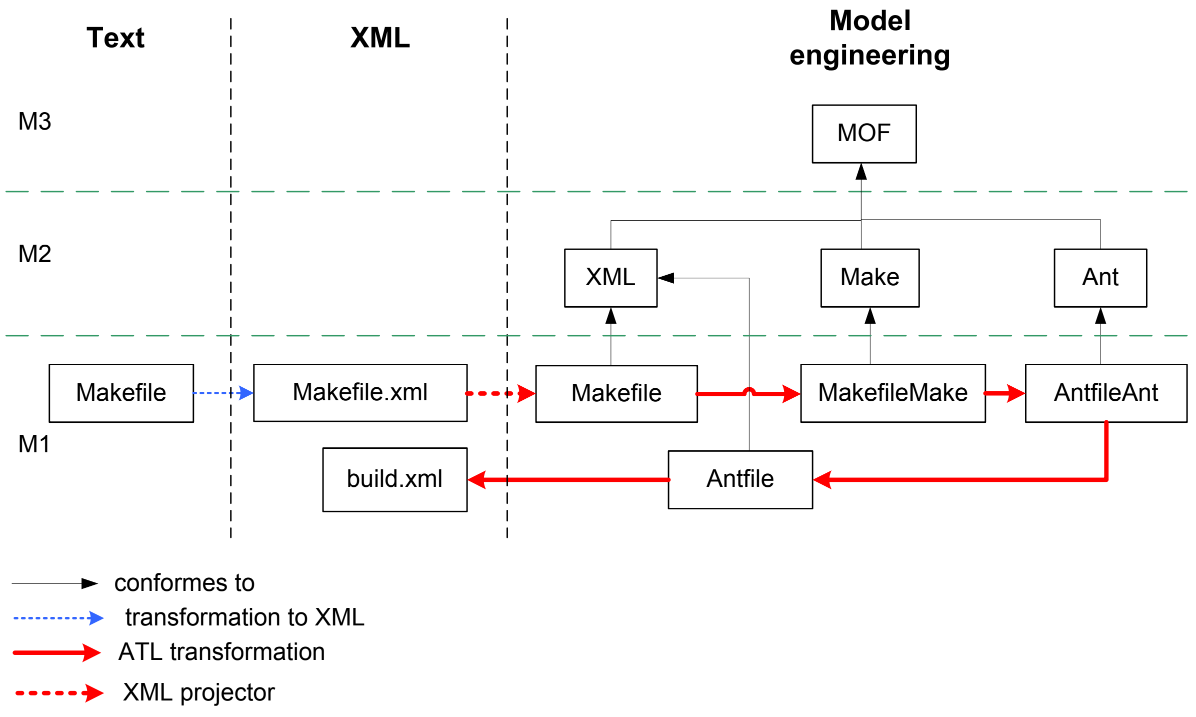

Figure 3. Transformation overview

This transformation is divided into several parts:

- the injector to obtain a file in file in xmi-format corresponding to the Make Metamodel;
- the transformation from the Make to the Ant Metamodel;
- the extractor to obtain a file in xml-format corresponding to Ant.

1.2. Metamodels

1.2.1. Make Metamodel

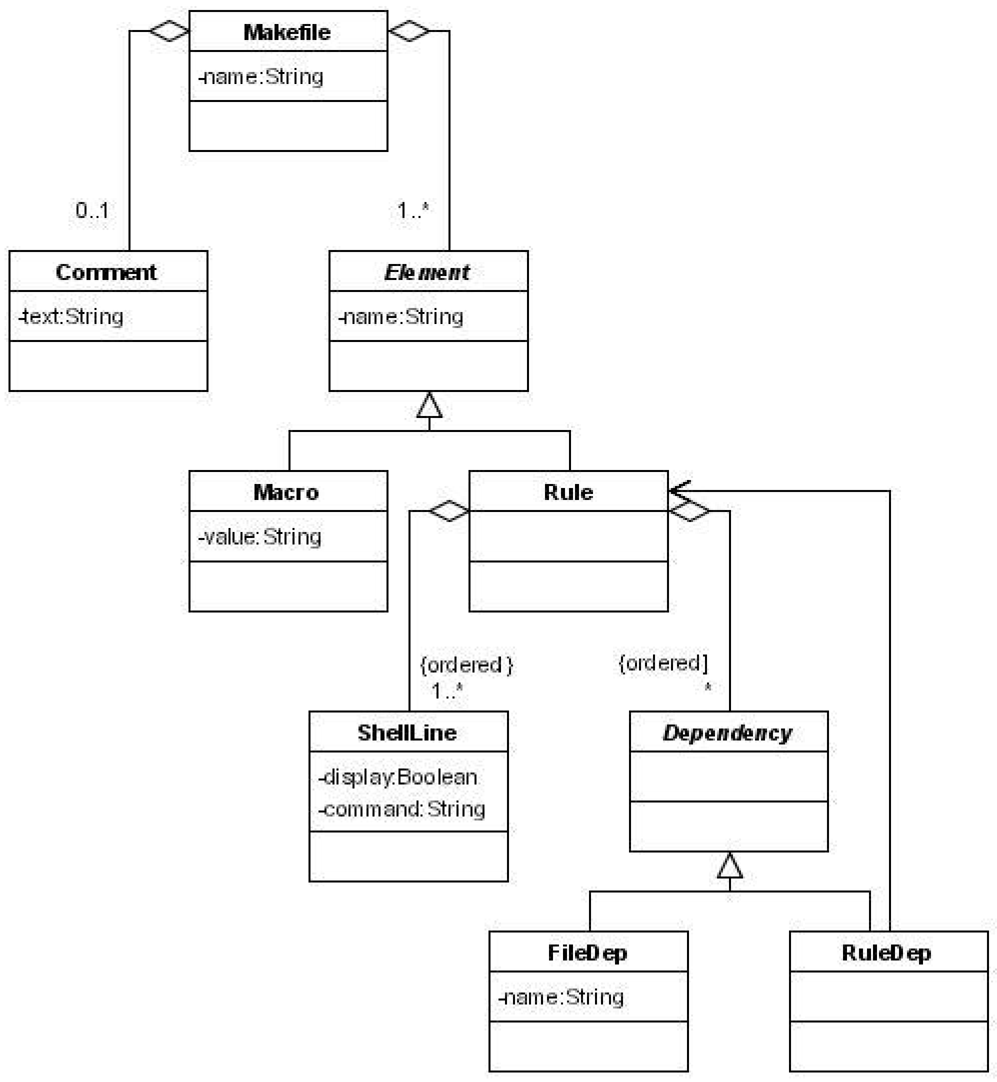

Figure 4. The Make metamodel

A Make is modelized by a Makefile element. This element contains several elements. An abstract Element can be either a Macro (to five a value to a constant) or a Rule.

A Macro has two attributes:

- the constant’s name;
- the constant’s value.

A Rule contains dependencies and shellLines (the commands which are called). There are two kinds of dependencies:

- the dependency of a file;

- the dependency of another rule.

A shellLine is a command which is called. When this command begins by the character '@', that means that there is no need to echo the command (the attribute *display* gets the value' false').

1.2.2. Ant Metamodel

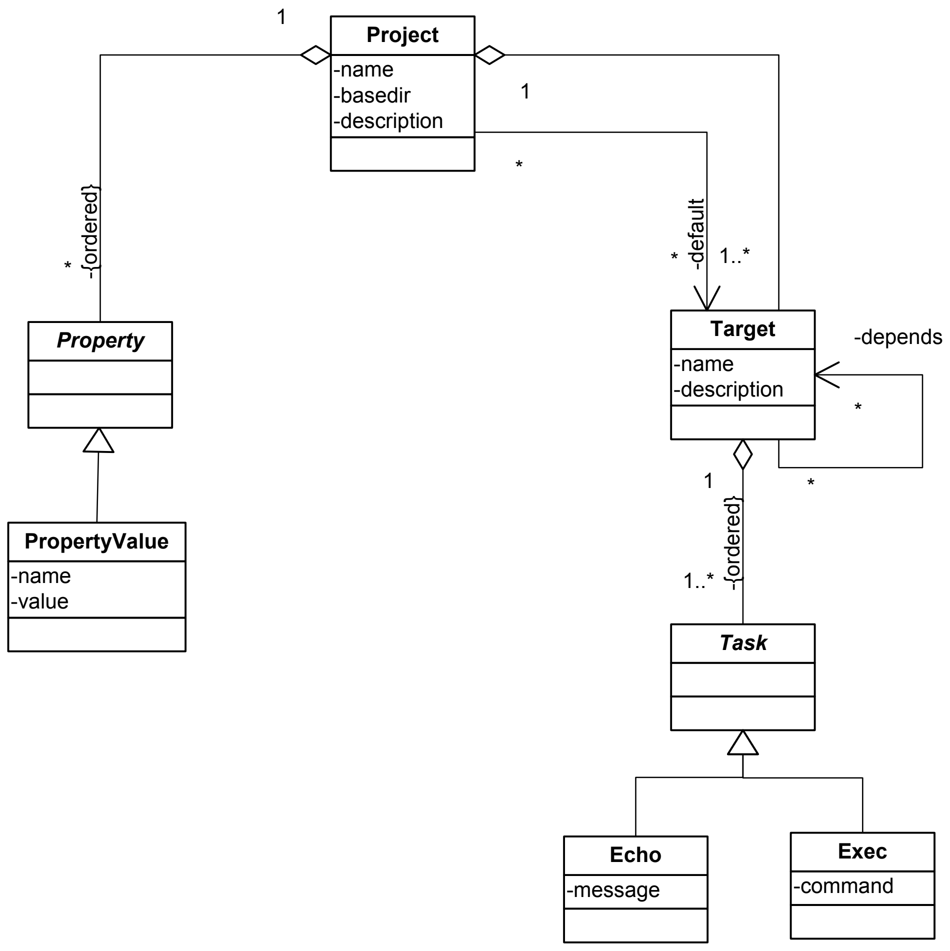

Figure 5. A simplified Ant Metamodel

An Ant file is modelized by a Project element. A project defines an ordered set of properties and one or more targets and which target is called by default.

A property serves to give a value to a constant.
A target is a set of tasks which must be executed after the execution of all its dependencies. For the make, there only two useful taks:
- echo, its role is to echo a message to the current loggers;
- exec, its role is to execute a system command.

1.3. Injector

The injector is divided in several parts:

- transformation from the makefile to a file in xml-format;
- importation of the xml model, to obtain a file in xmi-format corresponding to the XML Metamodel;
- transformation from a file corresponding to the XML Metamodel to a file corresponding to the Make Metamodel.

1.3.1. Transformation from the Makefile to a file in xml-format

The first transformation consists in creating a xml-based file from the makefile.

```
<make>
  <comment>Test d'un fichier makefile</comment>
  <macro name="CC" value="gcc"/>
  <macro name="CFLAGS" value="-wall -ansi"/>
  <macro name="LDFLAGS" value="-wall -ansi"/>
  <rule name="hello" depends="hello.o main.o">
    <shellLine>@$(CC) -o hello hello.o main.o $(LDFLAGS)</shellLine>
    <shellLine>skip</shellLine>
  </rule>
  <rule name="hello.o" depends="hello.c">
    <shellLine>@$(CC) -o hello.o -c hello.c $(CFLAGS)</shellLine>
  </rule>
  <rule name="main.o" depends="main.c hello.h">
    <shellLine>$(CC) -o main.o -c main.c $(CFLAGS)</shellLine>
  </rule>
  <rule name="clean">
    <shellLine>rm -rf *.o</shellLine>
  </rule>
  <rule name="mrproper" depends="clean">
    <shellLine>@rm -rf $(EXEC)</shellLine>
  </rule>
</make>
```

Figure 6. XML-based file representing a Makefile

To obtain this file, the command Unix is used: sed.

The XML-based file begins by the tag '<make>' and finished by the tag '</make>'.

The comments are recognized by the character '#' at the beginning of the line. All characters after '#' are copied without analysis:

```
s/#\(.*\)$/ <comment>\1<\/comment>/ p
```

The macros are recognized thanks to the character '='. The possible space and tab characters which are placed before or after the character '=' or after the value of the Macro are deleted. In this analysis, it can not have comments in the same line.

```
s/^\([a-zA-Z_][a-zA-Z0-9_]*\)[   ]*=[  ]*\(.*[A-Za-z0-9)]\)[   ]*$/  <macro     name="\1"
value="\2"\/>/ p
```

The rules are recognized by the character ':' and a shell command is after a tab character. The possible space and tab characters which are placed are placed before or after the character ':' or after the last dependency are deleted. But the spaces between two names of dependencies are not analysed. Concerning the shell command, all characters after the tab character are copied until the last letter (or number) which is found in this line. In this analysis, it can not have comments inside a rule.

```
    s/^\([A-Za-z][A-Za-z\.0-9]*\)[  ]*:[ ]*\(.*[A-Za-z0-9)]\)[   ]*$/ <rule
name="\1" depends="\2">/
      p
      n
      :s
      /^ /{
        s/^   //
        s/\(.*[A-Za-z0-9)]\)[   ]*$/    <shellLine>\1<\/shellLine>/
        p
        n
        bs
      }
      a\

      </rule>
```

The plug-in ‘org.atl.eclipse.km3importer_1.0.0’ creates a file in xmi.

1.3.2. Transformation from the XML Metamodel to the Make Metamodel

1.3.2.1. Rules Specification

These are the rules to transform a XML Model to a Make Model:

- For the Root, a Makefile element is created,
- For an Element which name is ‘comment’, a Comment element is created,
- For an Element which name is ‘macro’, a Macro element is created,
- Etc.

1.3.2.2. ATL Code

This ATL code for the XML to Make transformation consists of 7 helpers and 5 rules.

The getList helper is useful to extract the different names of dependencies for a rule. This list of names is given in parameter and a Sequence of String is returned. Two names of dependencies are separated by a space character. This helper uses another helper named getListAux.

The getAttribute helper is useful for all elements having Attribute children. Its rule consists in returning a value of an attribute whose name is given in parameter. It returns ‘’ if the required attribute does not exist. This helper uses testAttribute helper which indicates whether the attribute given in parameter exists, and getAttrVal helper which returns the value of an attribute.

The getText helper is useful for all element having Text children. Its rule consists in returning the value of a text whose name is given in parameter. In this helper, there is no test of existence.

The getShellLine helper allows extracting a shell command: that is to say to remove the first character if this one is the character ‘@’.

The rule Root2Makefile allocates a Makefile element.

The rule Comment allocates a Comment element. …

For the rule Rule, its dependencies can be represented by a FileDep or a RuleDep element. If it is a RuleDep element, a definition of this rule is present in this file:

```
   itsRuleDep : Sequence(XML!Element) = XML!Element.allInstances() ->
         select(d |  d.name = 'rule'
            and itsDependencies ->
                  includes( d.getAttribute('name')));
```

If it is a FileDep element, there is no rule having its name:

```
itsFileDep : Sequence(String) = itsDependencies ->
                         select(d | allRules ->
                             select(e|e.getAttribute('name')=d)->
                                      isEmpty()) ;
```

And the key words 'foreach […] distinct' are used to create a RuleDep (or FileDep) element for all elements present in the constant itsRuleDep (or itsFileDep).

1.4. Transformation from Make to Ant

1.4.1. Rules Specification

These are the rules to transform a Make model to an Ant model:

- For the MakeFile element, a Project in Ant element is created. As there is not equivalent of 'default' in a Make, this the first rule found in the makefile which will be called by default.
- For each Macro, a Property is created.
- For each Rule, a Target is created. Only the dependencies of rules are kept as dependencies.
- There are two kinds of ShellLines:
    - When the attribute *display* is true, the tags *Echo* and *Exec* are created,
    - When the attribute *display* is false, only the tag *Exec* is created.

1.4.2. ATL Code

This ATL code for the Make to Ant transformation consists of 5 rules.

The rule Makefile2Project allocates a Project element. This element is given a name and a description. It is linked to the Macro and the Rule contained in the Makefile.

The rule Macro2Property allocates a PropertyValue element. This new Property is given a name and a value.

The rule Rule2Target allocates a Target element. This new Target is given a name and it contains Tasks and it is linked to others targets.

The rule ShellLine2Task_Display allocates two Tasks: Echo and Exec. The Task Echo id given a message and the task Exec is given a command.

The rule ShellLine2Task_NoDisplay allocates the Task Exec.

1.5. Extractor

The extractor is divided into several parts:

- Transformation from a file corresponding to the Ant Metamodel to a file corresponding to the XML Metamodel,
- Creation of the XML file in Ant.

1.5.1. Transformation from Ant to XML Metamodel

1.5.1.1. Rules Specification

These are the rules to transform an Ant Model to a XML Model:

- For the Project, a Root element is created,
- For a Comment element, an Element which name is ‘comment’ is created,
- Etc.

1.5.1.2. ATL Code

This ATL code for the Ant to XML transformation consists of 1 helper and 24 rules.

The concat helper allows concatenating a sequence of string given in parameter. Two elements are separated by a comma. This helper is useful for the attribute depends of a target.

The rule Project2Root creates a Root element for the projects having an attribute named description:

```
rule Project2Root{
  from i : Ant!Project(
    if i.description.oclIsUndefined()
      then false
      else not(i.description='')
      endif
  )
  to o : XML!Root(…)
}
```

The ‘if then else’ instruction is used: when the first test failed, the second is not executed.

There is another rule Project2RootWithoutDescription for the project not having description. Thus, there is no Attribute element named ‘description’ which has no value.

There is a rule per element.

I. Make metamodel in KM3 format

```
package Make {
  --@begin central Class
  class Makefile {
     attribute name : String;
     reference comment[0-1] container : Comment;
     reference elements[1-*] ordered container : Element;
  }
  --@end central Class

  --@begin Elements
  --@comments represents differents elements contained in the makefile
  abstract class Element {
     attribute name : String;
  }

  --@comments represents a rule : a group of dependencies and of shellLines
  class Rule extends Element {
     reference dependencies[*] ordered container : Dependency;
     reference shellLines [1-*] ordered container : ShellLine oppositeOf
ruleShellLine;
  }

  --@comments represents a macro : to give a value
  class Macro extends Element {
     attribute value : String;
  }
  --@end Elements

  --@begin shellLine
  class ShellLine{
     attribute command : String;
     attribute display : Boolean;
     reference ruleShellLine : Rule oppositeOf shellLines;
  }
  --@end shellLine
  --@begin comment
  class Comment{
     attribute text : String;
  }
 --@end comment
 --@begin dependencies
 --@comments represents a dependency contained in a rule
 abstract class Dependency {
  }

  --@comments represents a dependency which call another rule
  class RuleDep extends Dependency {
     reference ruledep : Rule;
  }

  --@comments represents a file dependency
  class FileDep extends Dependency {
     attribute name : String;
  }
 --@end dependencies
}

package PrimitiveTypes{
    datatype String;
    datatype Boolean;
}
```

II. Ant metamodel in KM3 format (excerpt)

```
package Ant{

 class Project{
   attribute name : String;
   attribute description [0-1] : String;
   reference properties [*] ordered container : Property;
   reference targets [1-*] container : Target;
   reference "default" : Target;
 }

abstract class Property{}

abstract class PropertyName extends Property{
   attribute name : String;
}

class PropertyValue extends PropertyName{
   attribute value : String;
 }

 class Target{
   attribute name : String;
   reference depends [*] : Target;
   reference tasks [*] ordered container : PredefinedTask oppositeOf target;
 }

 abstract class Task{
 }
 abstract class PreDefinedTask extends Task{
   reference target : Target oppositeOf tasks;
 }

 abstract class ExecutionTask extends PreDefinedTask{}

 class Exec extends ExecutionTask{
   attribute executable : String;
   attribute dir : String;
 }

 abstract class MiscellaneousTask extends PreDefinedTask{}

 class Echo extends MiscellaneousTask{
   attribute message : String;
 }
}
package PrimitiveTypes{
   datatype String;
}
```

III. Make2Ant.sed file

```
#!/bin/sed -nf

# sed -nf make2xml.sed makefile >makefile.xml

1i\
<make>

:d
/\\$/{
   s/[  ]*\\$//
   N
   s/\n[   ]*/ /
   bd
}

# Get rid of comments
s/#\(.*\)$/  <comment>\1<\/comment>/ p

/^[a-zA-Z_][a-zA-Z0-9_]*[ ]*=/{
   s/^\([a-zA-Z_][a-zA-Z0-9_]*\)[  ]*=[ ]*\(.*[A-Za-z0-9)]\)[   ]*$/ <macro
name="\1" value="\2"\/>/  p
}

#rule with dependencies
/^[A-Za-z][A-Za-z\.0-9]*[ ]*:[ ]*\([A-Za-z].*\)$/{
   <rule name="\1">\n   <dependencies>\2<\/dependencies>/
   s/^\([A-Za-z][A-Za-z\.0-9]*\)[  ]*:[ ]*\(.*[A-Za-z0-9)]\)[   ]*$/ <rule name="\1"
depends="\2">/
   p
   n
   :s
   /^ /{
      s/^  //
      s/\(.*[A-Za-z0-9)]\)[   ]*$/    <shellLine>\1<\/shellLine>/
      p
      n
      bs
   }
   a\
   </rule>
}

#rule without dependencies
/^[A-Za-z][A-Za-z\.0-9]*[ ]*:[ ]$/{
   <rule name="\1">\n   <dependencies>\2<\/dependencies>/
   s/^\([A-Za-z][A-Za-z\.0-9]*\)[  ]*:[ ]*$/ <rule name="\1">/
   p
   n
   :s
   /^ /{
      s/^  //
      s/\(.*[A-Za-z0-9)]\)[   ]*$/    <shellLine>\1<\/shellLine>/
      p
      n
      bs
   }
   a\
   </rule>
```

```
}

$a\
</make>
```

IV. XML2Make.atl file

```
module XML2Make;
create OUT : Make from IN : XML;


-- -- to extract a list of String from a String -- --

-- helper getList: extract a sequence of String from the String listString
-- in the same order
-- (two elements are separated by a comma)
helper def:getList(listString : String):Sequence(String)=
   if(listString.size()=0)
      then Sequence{}
      else thisModule.getListAux(listString,1,1,Sequence{})
      endif;

-- helper getListAux
-- index1: begin of the word
-- index2: meter
helper def:getListAux(listString: String, index1: Integer, index2: Integer,
provSequence: Sequence(String)): Sequence(String)=
   if (listString.size()<index2)
      then provSequence -> append(listString.substring(index1,index2-1))
      else
         if listString.substring(index2,index2)=' '
            then thisModule.
                  getListAux(listString,index2+1,index2+1,provSequence ->
                        append(listString.substring(index1,index2-1)))
            else thisModule.
                  getListAux(listString,index1,index2+1, provSequence)
            endif
      endif;

-- -- to get an attribute -- --

-- helper getAttrVal: returns the value of the attribute 'name'
-- (without test of existence)
helper context XML!Element def: getAttrVal(name: String) : String =
    self.children ->
            select(c | c.oclIsKindOf(XML!Attribute) and c.name = name)->
                  first().value;

-- helper testAttribute: returns true if the attribute 'name' is defined
helper context XML!Element def: testAttribute(name : String) : Boolean =
   not (self.children ->
               select(d | d.oclIsKindOf(XML!Attribute) and d.name = name)->
                     first().oclIsUndefined());

-- helper: getAttribute: returns the value of the attribute
-- given in parameter
-- returns '' if this attribute does not exist
helper context XML!Element def:getAttribute(name : String):String =
   if (self.testAttribute(name))
      then self.getAttrVal(name)
      else ''
      endif;

-- -- others helpers -- --

-- helper getText:returns the value of a text belonging to an element
helper context XML!Element def: getText() : String =
    self.children ->
          select(c | c.oclIsKindOf(XML!Text)) ->
            first().value;

-- helper getShellLine: deletes the first character if this one is ‘@’
helper context XML!Element def : getShellLine():String=
  let thisText : String = self.getText() in
  if (thisText.substring(1,1)='@')
    then thisText.substring(2,thisText.size())
    else thisText
    endif;

-- rules

-- central rule
rule Root2MakeFile{
  from i : XML!Root
  to o : Make!Makefile(
    name <- 'makefile',
    comment <- i.children ->
          select(d | d.oclIsKindOf(XML!Element) and d.name = 'comment')->
                first(),
    elements <- i.children ->
          select(d | d.oclIsKindOf(XML!Element)
                and not (d.name = 'comment'))
  )
}

rule Comment{
  from i : XML!Element(
    i.name = 'comment'
  )
  to o : Make!Comment(
    text <- i.getText()
  )
}

rule Rule{
  from i : XML!Element(
    i.name = 'rule'
  )
  using{
  allRules : Sequence(XML!Element) = XML!Element.allInstances() ->
        select(d |  d.name = 'rule' );
  itsDependencies : Sequence(String) = thisModule.
        getList(i.getAttribute('depends'));
  itsRuleDep : Sequence(XML!Element) = XML!Element.allInstances() ->
      select(d |  d.name = 'rule'
          and itsDependencies ->
                includes( d.getAttribute('name')));
  itsFileDep : Sequence(String) = itsDependencies ->
        select(d | allRules -> select(e|e.getAttribute('name')=d)->
            isEmpty()) ;
  }
  to o : Make!Rule(
    name <- i.getAttribute('name'),
    dependencies <- Sequence{makeRuleDep,makeFileDep},
    shellLines <-  i.children ->
          select(d | d.oclIsKindOf(XML!Element) and d.name = 'shellLine')
  ),
      makeRuleDep : distinct Make!RuleDep foreach(dep in itsRuleDep)(
         ruledep <- dep
      ),
      makeFileDep : distinct Make!FileDep foreach(depFile in itsFileDep)(
         name <- depFile
      )
   }

   rule Macro{
      from i : XML!Element(
         i.name = 'macro'
      )
      to o : Make!Macro(
         name <- i.getAttribute('name'),
         value <- i.getAttribute('value')
      )
   }

   rule ShellLine{
      from i : XML!Element(
         i.name = 'shellLine'
      )
      to o  : Make!ShellLine(
         command <- i.getShellLine(),
         display <- not (i.getText().substring(1,1)='@')
      )
   }
```

V. Make2Ant.atl file

```
module Make2Ant;
create OUT : Ant from IN : Make;

-- rule Makefile2Project:  its the 'main' rule.
-- This rule generates the Project element.
-- Its attributes are the name and the description of the makefile.
-- It contains properties, targets of the makefile.
-- It defines also the target called by default.
rule Makefile2Project{
   from
      m : Make!Makefile
   to
      a : Ant!Project(
         name        <- m.name,
         description <- m.comment.text,
         properties <- m.elements ->
            select(c | c.oclIsKindOf(Make!Macro)),
         targets    <- m.elements ->
            select(c | c.oclIsKindOf(Make!Rule)),
         default    <- m.elements ->
            select(c | c.oclIsKindOf(Make!Rule)) -> first()
      )
}

-- rule Macro2Property:
-- This rule generates a Property.
-- Its attributes are the name and the value of the Macro.
rule Macro2Property{
   from
      m : Make!Macro
   to
      a : Ant!PropertyValue(
         name  <- m.name,
         value <- m.value
      )
}

-- rule Rule2Target:
-- This rule generates a Target.
-- Its attribute is the name of the rule.
-- It contains tasks.
-- It can be dependent of others targets.
rule Rule2Target{
   from
      m : Make!Rule
   to
      a:Ant!Target(
         name    <- m.name,
         tasks <- m.shellLines,
         depends <- m.dependencies ->
                     select(e | e.oclIsKindOf(Make!RuleDep)) ->
                        collect(e | e.ruledep)
      )
}

-- rule ShellLine2Task_Display
-- This rule is started when the attribute display is true.
-- This rule generate a Task Echo and a Task Exec.
```

```
rule ShellLine2Task_Display{
   from
      m : Make!ShellLine(
           m.display
         )
   to
      e:Ant!Echo(
         message <- m.command,
         target <- m.ruleShellLine
      ),
      x:Ant!Exec(
         executable <- m.command,
         target <- m.ruleShellLine
      )
}

-- rule ShellLine2Task_Display
-- This rule is started when the attribute display is false.
-- This rule generate only a Task Exec.
rule ShellLine2Task_NoDisplay{
   from
      m : Make!ShellLine(
           not m.display
         )
   to
      x:Ant!Exec(
         executable <- m.command,
         target <- m.ruleShellLine
      )
}
```

VI. Ant2XML.atl file

```
module Ant2XML;
create OUT : XML from IN : Ant;

-- concatene a list of String
-- the elements are separated by a comma
helper def: concat(list : Sequence(String)) : String =
   list -> asSet() -> iterate(element ;acc : String = '' |
                        acc +
                           if acc = ''
                           then element
                           else ',' + element
                        endif);

-- rule for a project having a description
rule Project2Root{
   from i : Ant!Project(
      if i.description.oclIsUndefined()
         then false
         else not(i.description='')
         endif
   )
   to o : XML!Root(
      name <- 'project',
      children <- Sequence {itsName,itsDescription,itsBasedir,
                            itsDefaultTarget,i.properties,
                            i.path,i.taskdef,i.targets}
   ),
    itsName : XML!Attribute(
      name <- 'name',
      value <- i.name
   ),
   itsDescription : XML!Element(
      name <- 'description',
      children <- textText
   ),
   textText : XML!Text(
      value <- i.description
   ),
    itsBasedir : XML!Attribute(
      name <- 'basedir',
      value <- i.basedir
   ),
   itsDefaultTarget : XML!Attribute(
      name <- 'default',
      value <- i.default.name
   )
}

-- rule for a project without description
rule Project2RootWithoutDescription{
   from i : Ant!Project(
      if i.description.oclIsUndefined()
         then true
         else i.description=''
         endif
   )
   to o : XML!Root(
      name <- 'project',
      children <- Sequence {itsName,itsBasedir,
                                 itsDefaultTarget,i.properties,
                                 i.path,i.taskdef,i.targets}
   ),
    itsName : XML!Attribute(
      name <- 'name',
      value <- i.name
   ),
    itsBasedir : XML!Attribute(
      name <- 'basedir',
      value <- i.basedir
   ),
   itsDefaultTarget : XML!Attribute(
      name <- 'default',
      value <- i.default.name
   )
}

-- properties

rule PropertyValue{
   from i : Ant!PropertyValue
   to o : XML!Element(
      name <- 'property',
      children <- Sequence{propertyName2,propertyValue}
   ),
   propertyName2 : XML!Attribute(
      name <- 'name',
      value <- i.name
   ),
   propertyValue : XML!Attribute(
      name <- 'value',
      value <- i.value
   )
}

-- …

-- target
rule TargetWithDescription{
   from i : Ant!Target(
      if i.description.oclIsUndefined()
         then false
         else not (i.description='')
         endif
   )
   to o : XML!Element(
      name <- 'target',
      children <- Sequence{nameAttribute,descriptionElement,
                                 dependsAttribute,i.tasks}
    ),
   nameAttribute : XML!Attribute(
      name <- 'name',
      value <- i.name
   ),
   descriptionElement : XML!Element(
      name <- 'description',
      children <- descriptionText
   ),
   descriptionText : XML!Text(
      value <- i.description
   ),
    dependsAttribute : XML!Attribute(
      name <- 'depends',
      value <- thisModule.concat(i.depends -> collect(e|e.name))
    )
  }

rule TargetWithoutDescription{
    from i : Ant!Target(
      if i.description.oclIsUndefined()
        then true
        else i.description=''
        endif
    )
    to o : XML!Element(
      name <- 'target',
      children <- Sequence{nameAttribute,dependsAttribute,i.tasks}
    ),
    nameAttribute : XML!Attribute(
      name <- 'name',
      value <- i.name
    ),
    dependsAttribute : XML!Attribute(
      name <- 'depends',
      value <- thisModule.concat(i.depends -> collect(e|e.name))
    )
  }

-- tasks
-- …


-- pre-defined tasks
-- …

rule Exec{
    from i : Ant!Exec
    to o : XML!Element(
      name <- 'exec',
      children <- execAttribute
    ),
    execAttribute : XML!Attribute(
      name <- 'executable',
      value <- i.executable
    )
}

rule Echo{
    from i : Ant!Echo
    to o : XML!Element(
      name <- 'echo',
      children <- echoAttribute
    ),
    echoAttribute : XML!Attribute(
      name <- 'message',
      value <- i.message
    )
}
-- …
-- it exists others rules in this file, but they are never called for this --
project.
```

VII. XML2Text.atl file

```
query XML2Text = XML!Root.allInstances()
    ->asSequence()
    ->first().toString2('').writeTo('MyDirectory\\project.xml');


helper context XML!Element def: toString2(indent : String) : String =
  let na : Sequence(XML!Node) =
    self.children->select(e | not e.oclIsKindOf(XML!Attribute)) in
  let a : Sequence(XML!Node) =
    self.children->select(e | e.oclIsKindOf(XML!Attribute)) in
  indent + '<' + self.name +
  a->iterate(e; acc : String = '' |
    acc + ' ' + e.toString2()
  ) +
  if na->size() > 0 then
    '>'
    + na->iterate(e; acc : String = '' |
      acc +
      if e.oclIsKindOf(XML!Text) then
        ''
      else
        '\r\n'
      endif
      + e.toString2(indent + '  ')
    ) +
    if na->first().oclIsKindOf(XML!Text) then
      '</' + self.name + '>'
      else
        '\r\n' + indent + '</' + self.name + '>'
    endif
  else
    '/>'
  endif;


helper context XML!Attribute def: toString2() : String =
  if (self.value.oclIsUndefined())or(self.value='')
    then ''
    else self.name + '=\"' + self.value + '\"'
    endif;
--self.name + '=\"' + self.value + '\"';

helper context XML!Text def: toString2() : String =
  self.value;
```

References

[1] Make Description. http://www.gnu.org/software/make/

[2] Overview of Ant. http://ant.apache.org/manual/

[3] KM3: Kernel MetaMetaModel. Available at http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/doc/atl/index.html.