1. ATL Transformation Example

1.1. Example: Monitor → Semaphore

The Monitor to Semaphore example describes a transformation from a program containing Hoare's monitors [1] definitions into an equivalent program in which synchronization operations are based on Dijkstra's semaphores [2]. This is an example of end-to-end transformation, from code to code. It strictly follows the operational semantic of monitors described in terms of semaphores and shows how this abstract description may be implemented into an executable ATL transformation.

Both input and output transformation models are based on the "program" metamodel. This metamodel describes a set of common pseudocode instructions (variables and procedures declarations, procedures calls, monitors definitions, etc.).

The way a monitor-based program can be transformed into a semaphore-based program has been described by Hoare [1]. This kind of transformation can be summarized by the following points:

- Monitors are removed. Three variables (mutex, urgent and urgent count) are now used to manage each monitor.
- Conditions are removed. Two variables (condsem and count) are now used to manage each condition (of each monitor). Two procedures are also generated for each condition to be substituted to the "signal" and "wait" condition procedures. They are detailed in Table 1.

```
procedure wait() {
   condcount := condcount + 1;
   if urgentcount > 0 then {
      urgent.v();
   }
   else {
      mutex.v();
   }
   condsem.p();
   condcount := condcount - 1;
}
```

```
procedure signal() {
   urgentcount := urgentcount + 1;
   if condcount > 0 then {
      condsem.v();
      urgent.p()
   }
   urgentcount := urgentcount - 1;
}
```

Table 1. Pseudocode for generated wait and signal procedures

- Since only one program at a time is allowed to enter a monitor procedure, entrance and exit of monitors' procedures now have to be managed by a dedicated code which is inserted at the beginning and the end of each monitor procedure. Details of this code are provided in Table 2.

```
procedure_entrance code:
   mutex.p();
```

```
procedure_exit code:
   if urgentcount > 0 then {
      urgent.v();
   }
   else {
      mutex.v();
   }
```

Table 2. Pseudocode for generated wait and signal procedures

1.2. An example

The example developed in this section corresponds to the SingleResource example introduced by Hoare in its paper [1]. The SingleResource program is presented in Figure 1.

```
program SingleResourceProgram {

   monitor SingleResource {
      var busy : boolean := false;
      var nonBusy : condition;

      procedure acquire() {
         if busy then {
            nonBusy.wait();
         } else {
            busy := true;
         }
      }

      procedure release() {
         busy := false;
         nonBusy.signal();
      }
   }
}
```

Figure 1. Monitor-based SingleResourceProgram

The SingleResource program only contains a monitor definition. This monitor includes declarations of a condition (nonBusy) and a boolean variable (busy), as well as two procedures definitions (acquire and release). The transformed version of this program is provided in Figure 2.

In this new version, the monitor has been removed, and replaced by standalone variables and procedures. Thus, the two monitors' procedures are now standalone procedures (SingleResource_acquire and SingleResource_release). The monitor boolean variable has also been associated with a standalone variable (SingleResource_busy). Monitor management is now insured by means of three new variables: SingleResource_mutex, SingleResource_urgent and SingleResource_urgentcount.

Finally, the transformation of the condition has lead to the apparition of two additional variables (SingleResource_nonBusy_sem and SingleResource_nonBusy_count), whereas the "signal" and "wait" condition procedure calls have been substituted by two standalone procedures (SingleResource_nonBusy_signal and SingleResource_nonBusy_wait).

```
program SingleResourceProgram {

   var SingleResource_nonBusy_sem : boolean := false;
   var SingleResource_nonBusy_count : integer := 0;
   var SingleResource_mutex : boolean := true;
   var SingleResource_urgent : boolean := false;
   var SingleResource_urgentcount : integer := 0;
   var SingleResource_busy : boolean := false;

   procedure SingleResource_acquire() {
      SingleResource_mutex.p();
      if SingleResource_busy then {
         this.SingleResource_nonBusy_wait();
      } else {
         SingleResource_busy := true;
      }
      if SingleResource_urgentcount > 0 then {
         SingleResource_urgent.v();
      } else {
         SingleResource_mutex.v();
      }
   }

   procedure SingleResource_release() {
      SingleResource_mutex.p();
      SingleResource_busy := false;
      this.SingleResource_nonBusy_signal();
      if SingleResource_urgentcount > 0 then {
         SingleResource_urgent.v();
      } else {
         SingleResource_mutex.v();
      }
   }

   procedure SingleResource_nonBusy_wait() {
      SingleResource_nonBusy_count := SingleResource_nonBusy_count + 1;
      if SingleResource_urgentcount > 0 then {
         SingleResource_urgent.v();
      } else {
         SingleResource_mutex.v();
      }
      SingleResource_nonBusy_sem.p();
      SingleResource_nonBusy_count := SingleResource_nonBusy_count - 1;
   }

   procedure SingleResource_nonBusy_signal() {
      SingleResource_urgentcount := SingleResource_urgentcount + 1;
      if SingleResource_nonBusy_count > 0 then {
         SingleResource_nonBusy_sem.v();
         SingleResource_urgent.p();
      }
      SingleResource_urgentcount := SingleResource_urgentcount - 1;
   }
}
```

Figure 2. Semaphore-based SingleResourceProgram

1.3. Metamodel

The Monitor to Semaphore transformation is based on a metamodel that provides a set of peudocode instructions. Available instructions make it possible to specify programs including both the definition of monitors and some code making use of the defined monitors. The considered metamodel is presented in Figure 3.

A Program inherits both from Structure and ProcContainerElement. A Progam can contain VariableDeclarations (as a Structure), Procedures (as a ProcContainerElement), and Monitors. A Monitor is also a Structure and a ProcContainerElement, and can therefore contain VariableDeclarations and Procedures. Besides Statements and Parameters, a Procedure, as a Structure, can also contain VariableDeclarations.

Each VariableDeclaration is associated with a one and only Type. It may also contain an initial value that is represented by an Expression (see below). Parameters inherit from VariableDeclaration. They are characterized by a "direction" attribute ("in" or "out").

A Procedure contains a sequence of Statements. The Program metamodel provides support for the following statement types:

- An AssignmentStat contains a "target" VariableExp and a "value" Expression.
- A WhileStat contains a "condition" Expression and several "doStats" Statements.
- A ConditionalStat contains a "condition" Expression, several "thenStats" Statements and, optionally, "elseStats" Statements.
- Finally, an ExpressionStat simply contains an Expression.

Expression is an abstract entity from which the following elements inherit:

- IntegerExp and BooleanExp (which inherit from the abstract LiteralExp entity).
- VariableExp, which is associated with a VariableDeclaration.
- PropertyCallExp (abstract), which is characterized by its "source" element (of type Expression).

There exist three types of PropertyCallExp: the AttributeCallExp, the OperatorCallExp and the ProcedureCallExp. An OperatorCallExp contains a right element (of type Expression). A ProcedureCallExp can contain "argument" Expressions.

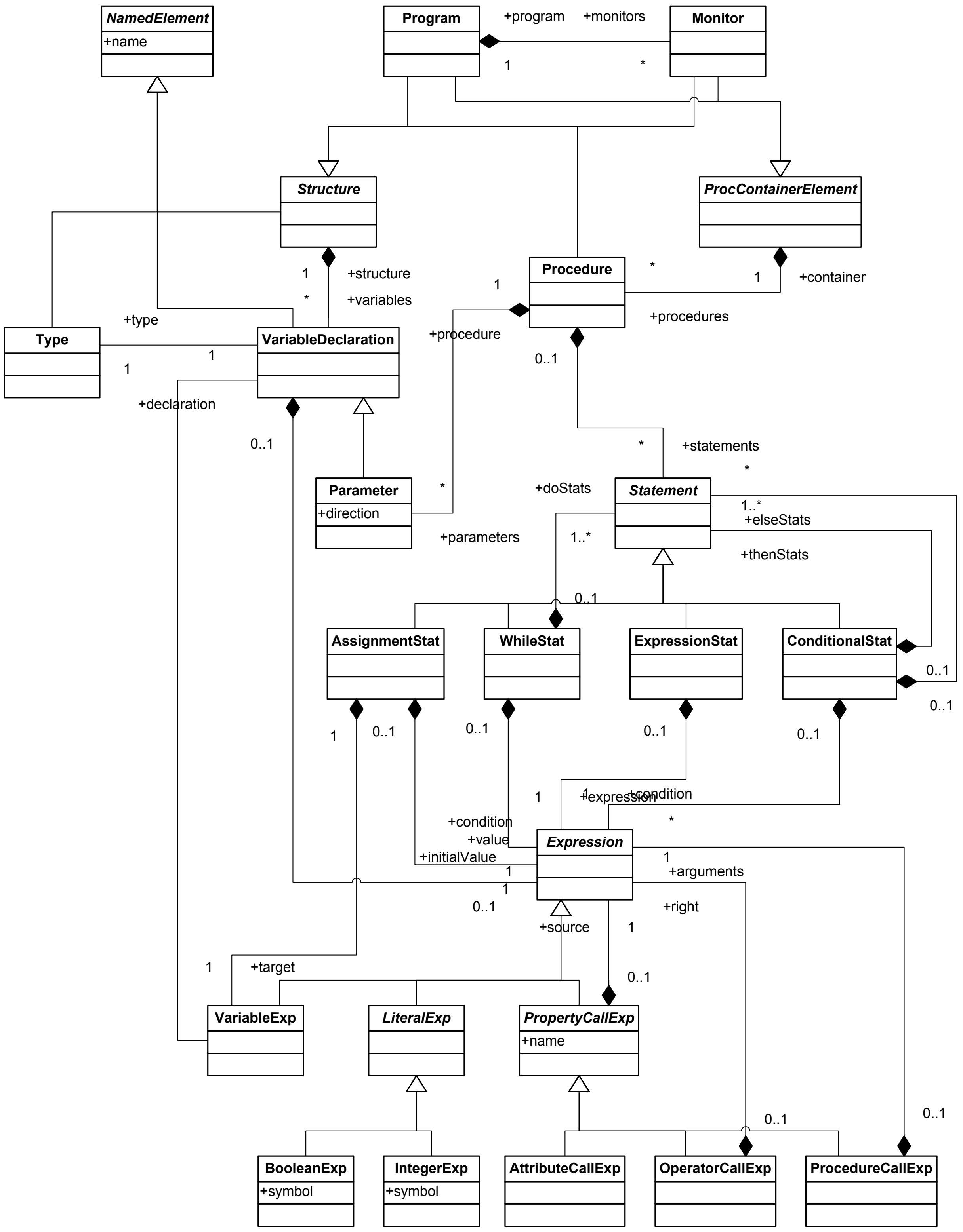

Figure 3. The Program metamodel

1.4. Rules Specification

Here are the main rules to transform a Monitor-based program into a Semaphore-based one:

- For the root **Program** element, a Monitor-free Program is generated. Two Program transformation rules are specified in order to handle Program with or without defined Monitors:
    - In case the Program contains Monitors definitions, variables of integer and boolean types have to be generated (see the Monitor rule). Since these types may be undefined in the input Program, they are generated at this point. In case they are also defined in the input Program, they are not generated twice by the transformation program (see the Type rule).
    - When no Monitors are defined, a corresponding Program is generated.
- For each **Type** element which is different from “condition”, a corresponding Type is generated if either:
    - No Monitors are defined in the input Program.
    - Or Monitors are defined in the input Program, but the type is different from “integer” and “boolean” (in this case, these Types are already generated by the Program rule).
- For each **Monitor**, three VariableDeclarations are created. Several Monitors may be defined in a single program. Since all generated VariableDeclarations have to be attached to the main Program, their names have to be prefixed by the name of their respective Monitor. Here are the generated elements:
    - A “mutex” VariableDeclaration (of type boolean, initialized to true).
    - An “urgent” VariableDeclaration (of type boolean, initialized to false).
    - An “urgentcount” VariableDeclaration (of type integer, initialized to 0).
    - Three LiteralExp are also generated for variables initialization.
- For each standalone **Procedure** (not declared within a Monitor definition), a corresponding Procedure is generated.
- For each **Procedure** declared within a Monitor definition, a Procedure, which name is prefixed by the monitor name, is generated. Statements of the generated Procedure are preceded by a Monitor entering statement (“P(mutex)”), and followed by a Monitor leaving statement (“if urgentcount > 0 then V(urgent) else V(mutex)”). Thus, generation of the Monitor entering statement includes the generation of:
    - An ExpressionStat.
    - A ProcedureCallExp (named “p”).
    - A VariableExp (pointing to the “mutex” variable generated for the Monitor).
- For each **VariableDeclaration**, which is not a Parameter, and which type is different from “condition”, a VariableDeclaration is generated. However, due to the management of Type elements (that may be generated from different rules), four cases have to be handled:
    - The variable is directly (declared within a Monitor) or indirectly (declared within a Procedure of a Monitor) part of a Monitor, and its type is “integer” or “boolean”. The name of the generated variable is prefixed by the Monitor name. Its type points to the

corresponding type generated by the Program rule. If the input variable is directly declared in a Monitor, the generated variable is attached to the main Program. If the input variable is declared in a Procedure of a Monitor, the generated variable is attached to the corresponding generated Procedure.

  - The variable is part of a Monitor, but its type is different from “integer” and “boolean”. The name of the generated variable is prefixed by the Monitor name, but its type points to the one of the input VariableDeclaration. If the input variable is directly declared in a Monitor, the generated variable is attached to the main Program. If the input variable is declared in a Procedure of a Monitor, the generated variable is attached to the corresponding generated Procedure.
  - The variable is not part of a Monitor, and its type is “integer” or “boolean”. The type of the generated variable points to the corresponding type generated by the Program rule.
  - The variable is not part of a Monitor, but its type is different from “integer” and “boolean”. The type of the generated variable points to the one of the input VariableDeclaration.

- For each **Parameter**, a Parameter is generated. If 1) Monitors are defined within the input Program, and 2) the Parameter is of type “integer” or “boolean”, the type of the generated Parameter points to the corresponding type generated by the Program rule. Otherwise, it points to the one type of the input Parameter.
- For each “condition” **VariableDeclaration**, four main elements (two variables and two procedures) are generated. The name of the generated elements is prefixed by both the Monitor and the condition names.
  - A “condsem” variable (of type boolean, initialized to false).
  - A “count” variable (of type integer, initialized to 0).
  - A “wait” procedure composed of four statements (see Section 1.5 for further details).
  - A “signal” procedure composed of three statements (see Section 1.5 for further details).
- For each “condition” **ProcedureCallExp** (a ProcedureCallExp which source type is “condition”), the two following elements are generated:
  - A VariableExp which points to the “this” VariableDeclaration generated in the Program transformation rule
  - A ProcedureCallExp which name is prefixed by both the Monitor and “condition” names, and which source is associated to the previously generated VariableExp.
- For each “non-condition” **ProcedureCallExp** (a ProcedureCallExp which source type is not “condition”), a corresponding ProcedureCallExp is generated.
- For each **VariableExp** that is not associated with a “condition” VariableDeclaration, a corresponding VariableExp is generated.

1.5. ATL Code

This ATL code for the Monitor to Semaphore transformation consists of 10 helpers and 24 rules.

1.5.1. Helpers

Five of the defined helpers are constant helpers. Among these, the **rootElt** helper provides access to the unique Program root element. The **allCondVariables** helper computes the set of "condition" VariableDeclaration. In the same way, the **allMonitors** helper computes the Monitors set. The **monitorsDefined** helper computes a boolean value indicating whether monitors (at least one) are defined in the input Program. The **allCondCalls** helper computes the set of procedure calls VariableExp of "condition" type.

The **getTargetStructure()** helper computes the Structure a VariableDeclaration declared in a Monitor (or in a Monitor Procedure) has to be attached to. If the variable is directly declared in a Monitor, it is attached to the main Program, otherwise, it is attached to its Procedure.

When Monitors are defined in the input Program, the **getType()** helper provides access, for a VariableDeclaration of type "integer" or "boolean", to its corresponding Type element generated in the MainWithMonitor rule.

The **isDeclaredInMonitor()** helper indicates whether or not a VariableDeclarartion is directly or indirectly (i.e. within a Monitor Procedure) declared in a Monitor.

When Monitors are specified in the input Program model, the **getProcedures()** helper computes the set of Procedures to be attached to the root Program element. This set is composed of the existing standalone Procedures, associated with the Procedures defined in the Monitors, and the Procedures generated for each condition (the "wait" and "signal" Procedures).

Finally, when Monitors are specified in the input Program model, the **getVariables()** helper computes the set of VariableDeclarations to be attached to the root Program element. This set is composed of the existing standalone VariableDeclarations, associated with the VariableDeclarations defined in the Monitors and the generated VariableDeclarations ("mutex", "urgent" and "urgentcount" for each Monitor, and "condsem" and "cond" for each condition).

1.5.2. Rules

Besides helpers, the Monitor to Semaphore transformation is composed of 24 rules.

When Monitors are defined within the input Program model, the **MainWithMonitors** rule aims to generate an output Program element. The generated Program is associated with an empty monitor set, and with the variables and procedures respectively computed by the getVariables() and getProcedures() helpers. Moreover, the rule also generates the "boolean" and "integer" Types elements, as well as the "this" global VariableDeclaration (of the "program" Type).

When no Monitors are defined within the input Program model, the **MainWithoutMonitors** rule generates an output Program element by simply copying the input Program properties.

The **Monitor** rule generates three VariableDeclarations that are associated with each declared Monitor in the source Program: the "mutex", "urgent" and "urgentcount" variables. Their names are prefixed by the name of the Monitor. Their respective types point to the corresponding Type generated by the MainWithMonitor rule. Three LiertalExp are also generated by the rule in order to be associated with the variables' initial values.

For each VariableDeclaration that belongs to allCondVariables, the **Condition** rule generates four main elements: the "condsem" and "count" VariableDeclarations, and the "wait" and "signal" Procedures. Names of both generated variables and procedures are prefixed by the names of the Monitor and the condition. All of them are attached to the root Program element.

The "condsem" variable type points to the "boolean" Type generated by MainWithMonitors rule. Its initialValue points to a "false" BooleanExp also generated by the Condition rule. The "count" variable type points to the "integer" Type generated by MainWithMonitors rule. Its initialValue points to a "0" IntegerExp also generated by the Condition rule.

The "wait" Procedure is composed of a sequence of four statements which require generating more than twenty elements. We provide here an overview of the generation of these statements:

- First statement is an AssignmentStat, which contains a VariableExp (pointing to the locally generated "count" VariableDeclaration) as target, and an OperatorCallExp as value property. This operator has "+" as name, contains another VariableExp (also pointing to the locally generated "count" VariableDeclaration) as target, and has a "1" IntegerExp as right property.

- Second statement is a ConditionalStat composed of an OperatorCallExp as test, a "then" statement and a "else" statement. The test OperatorCallExp has ">" as name, a VariableExp (pointing to the locally generated "count" VariableDeclaration) as source, and a "0" IntegerExp as right property. The "then" statement is an ExpressionStat containing an argument-less ProcedureCallExp. Its name is "v", and its source is a VariableExp pointing to "urgent" VariableDeclaration generated by the Monitor rule. Finally, the "else" statement is an ExpressionStat also containing an argument-less ProcedureCallExp. The name of this procedure call is "v", and its source is a VariableExp pointing to "mutex" VariableDeclaration generated by the Monitor rule.

- Third statement is an ExpressionStat that contains a ProcedureCallExp. This expression has "p" as name, has no arguments, and contains a VariableExp (pointing to the locally generated "condsem" VariableDeclaration) as source.

- Fourth statement is similar to the first one, except that the OperatorCallExp generated in this scope has "-" as name.

Regarding the generation of the "signal" Procedure, which is based on the same principles, the reader is invited to refer to the ATL code at the end of this section.

The **StandaloneProcedure** rule generates a Procedure for each Procedure directly declared in the input Program by simply copying the input Procedure properties.

The **MonitorProcedure** rule generates a Procedure for each Procedure declared in a Monitor. The name of the generated Procedure is prefixed by the Monitor name. It is attached to the root Program. Its Parameters and variables are copied from the input Procedure, but its body (that is the statements) is inserted between two statements which aim is to manage concurrency for Monitor accesses. The Procedure entering statement corresponds to "P(mutex)". It requires generating three elements:

- The entering ExpressionStat that contains the expression.

- The ProcedureCallExp, which has no parameter, which name is "p", and which source is the "mutex" VariableExp.

- The VariableExp which points to the "mutex" VariableDeclaration generated by the Monitor rule.

The Procedure leaving statement corresponds to “if urgentcount > 0 then V(urgent) else V(mutex)”. This ConditionalStat is itself composed of a “test” Expression, a “then” and an “else” statements. It requires the generation of the following elements:

- The “test” OperatorCallExp, with “>” as operator name, a VariableExp (pointing to the “urgentcount” VariableDeclaration generated by the Monitor rule) as source, and an IntegerExp.
- The “then” ExpressionStat which contains a ProcedureCallExp, which name is “v”, with no arguments, and which source is a VariableExp pointing to the “urgent” VariableDeclaration generated by the Monitor rule.
- The “else” ExpressionStat which also contains a ProcedureCallExp, which name is “v”, with no arguments, and which source is a VariableExp pointing to the “mutex” VariableDeclaration generated by the Monitor rule.

The **Type** rule generates a Type (by copying its name) for input Types which name is different from “condition” in the following cases:

- No Monitors are defined in the input Program.
- Monitors are defined in the input Program, but the type is different from “integer” and “boolean” (since, in the case of Monitors definition, these two types are already generated in the MainWithMonitors rule).

The **BooleanIntegerMonitorVariableDeclaration** rule generates a VariableDeclaration for input declarations of type “integer” or “boolean” that are declared within a Monitor, and that are not Parameters. The name of the generated variable is prefixed by the Monitor name. Its type is computed by the getType() helper, while the structure it is attached to is computed by the getTargetStructure() one.

The **MonitorVariableDeclaration** rule generates a VariableDeclaration for input declarations which types are different from “condition”, “integer” and “boolean”, that are declared within a Monitor, and that are not Parameters. The name of the generated variable is prefixed by the Monitor name. The structure the generated declaration is attached to is computed by the getTargetStructure() helper. Its type is copied from the input VariableDeclaration one.

The **BooleanIntegerStandaloneVariableDeclaration** rule generates a VariableDeclaration for input declarations of type “integer” or “boolean” that are not declared within a Monitor, and that are not Parameters. The name of the generated variable is prefixed by the Monitor name. Its type is computed by the getType() helper, while the structure it is attached to is copied from the input VariableDeclaration one.

The **StandaloneVariableDeclaration** rule generates a VariableDeclaration for input declarations which types are different from “condition”, “integer” and “boolean”, that are not declared within a Monitor, and that are not Parameters. The name of the generated variable is prefixed by the Monitor name. Both structure and Type of the generated declaration are copied from the input VariableDeclaration values.

For each parameter of Type “integer” or “boolean” declared in the scope of a procedure defined within a Monitor, the **BooleanIntegerMonitorParameter** rule generates a Parameter. The type of the generated Parameter is computed by the getType() helper, while its other properties are copied from the input Parameter values.

The **Parameter** rule generates a Parameter for all the Parameters that are not handled by the previous rule (BooleanIntegerMonitorParameter). All the properties of the generated Parameter are copied from the input Parameter values.

The **ConditionProcCallExp** rule generates a ProcedureCallExp for all ProcedureCallExp which declaration type is “condition”. The name of the called condition Procedure is copied from the input ConditionProcCallExp and prefixed by both the condition Monitor and the condition names. The source of the Procedure call is associated to the global “this” variable. For this purpose, a VariableExp is also generated that points to the “this” VariableDeclaration generated in the MainWithMonitors rule.

The ten remaining rules only copy their input element to their output element. Further details can be found in the transformation code below.

```
module Monitor2Semaphore;
create OUT : Program from IN : Program;


-------------------------------------------------------------------------------
-- HELPERS --------------------------------------------------------------------
-------------------------------------------------------------------------------

-- This helper provides access to the root Program element.
-- CONTEXT: thisModule
-- RETURN: Program!Program
helper def: rootElt : Program!Program =
   Program!Program.allInstances()->asSequence()->first();

-- This helper builds the set of 'condition' VariableDeclaration elements.
-- CONTEXT: thisModule
-- RETURN: Set(Program!VariableDeclaration)
helper def: allCondVariables : Set(Program!VariableDeclaration) =
   Program!VariableDeclaration.allInstances()
      ->select(c | c.type.name = 'condition');

-- This helper builds the set of Monitor elements.
-- CONTEXT: thisModule
-- RETURN: Set(Program!Monitor)
helper def: allMonitors : Set(Program!Monitor) =
   Program!Monitor.allInstances();

-- This helper provides a boolean indicating if monitors (at least one) are
-- defined in the input model.
-- CONTEXT: thisModule
-- RETURN: Boolean
helper def: monitorsDefined : Boolean =
   thisModule.allMonitors->notEmpty();

-- This helper builds the set of 'condition' calls VariableExp elements.
-- CONTEXT: thisModule
-- RETURN: Set(Program!VariableExp)
helper def: allCondCalls : Set(Program!VariableExp) =
   Program!ProcedureCallExp.allInstances()->collect(e | e.source);
--    ->collect(e | allCondVariables->includes(e.source.declaration));

-- This helper computes the Structure (Program, Monitor, Procedure) element
-- the context VariableDeclaration has to be attached to:
--   * a standalone declaration is attached to the Program element;
--   * a monitor declaration is attached to the Program element;
--   * an in-procedure declaration is attached to its Procedure element.
-- CONTEXT: Program!VariableDeclaration
-- RETURN: Program!Structure
helper context Program!VariableDeclaration
   def: getTargetStructure() : Program!Structure =
   let v_container : Program!Structure = self.structure in
   if v_container.oclIsKindOf(Program!Monitor) then
      v_container.program
   else
      v_container
   endif;

-- This helper returns the Type of a VariableDeclaration (which may be a
-- Parameter) when:
--   1- this variable is of 'integer' or 'boolean' type;
--   2- at least 1 monitor has been defined in the input program.
-- CONTEXT: Program!VariableDeclaration
-- RETURN: Program!Type
helper context Program!VariableDeclaration
   def: getType() : Program!Type =
   if self.type.name = 'integer' then
      thisModule.resolveTemp(thisModule.rootElt, 'integer_type')
   else
      thisModule.resolveTemp(thisModule.rootElt, 'boolean_type')
   endif;

-- This helper computes a boolean indicating whether the context variable is
-- declared within a monitor or a procedure of a monitor.
-- CONTEXT: Program!VariableDeclaration
-- RETURN: Boolean
helper context Program!VariableDeclaration
   def: isDeclaredInMonitor() : Boolean =
   let v_container : Program!Structure = self.structure in
   if v_container.oclIsKindOf(Program!Monitor) then
      true
   else
      if v_container.oclIsKindOf(Program!Procedure) then
         v_container.container.oclIsKindOf(Program!Monitor)
      else
         false
      endif
   endif;

-- This helper computes the set of Procedure elements to be attached to the
-- root Program element. This set includes:
--   * the defined standalone procedures;
--   * the 'signal', and 'wait' procedures generated for each input Monitor
--     element;
--   * the 'cond_wait' and 'cond_signal' procedures generated for each input
--     'condition' element.
-- CONTEXT: Program!Program
-- RETURN: Set(Program!Procedure)
helper context Program!Program
   def: getProcedures() : Set(Program!Procedure) =
   self.monitors->collect(e | e.procedures)
      ->union(thisModule.allCondVariables
         ->collect(e | thisModule.resolveTemp(e, 'cond_wait')))
      ->union(thisModule.allCondVariables
         ->collect(e | thisModule.resolveTemp(e, 'cond_signal')))
      ->union(self.procedures);

-- This helper computes the set of VariableDeclaration elements to be attached
-- to the root Program element. This set includes:
--   * the defined standalone variables;
--   * the 'mutex', 'urgent' and 'urgentcount' variables generated for each
--     input Monitor element;
--   * the 'condsem' and 'count' variables generated for each input
--     'condition' element.
-- CONTEXT: Program!Program
-- RETURN: Set(Program!VariableDeclaration)
helper context Program!Program
   def: getVariables() : Set(Program!VariableDeclaration) =
   self.variables
      ->union(thisModule.allCondVariables
         ->collect(e | thisModule.resolveTemp(e, 'condsem')))
      ->union(thisModule.allCondVariables
         ->collect(e | thisModule.resolveTemp(e, 'count')))
      ->union(thisModule.allMonitors
         ->collect(e | thisModule.resolveTemp(e, 'mutex')))
      ->union(thisModule.allMonitors
         ->collect(e | thisModule.resolveTemp(e, 'urgent')))
      ->union(thisModule.allMonitors
         ->collect(e | thisModule.resolveTemp(e, 'urgentcount')));



-------------------------------------------------------------------------------
-- RULES ----------------------------------------------------------------------
-------------------------------------------------------------------------------

-- Rule 'MainWithMonitors'.
-- This rule generates the structure of the root Program element when the
-- input element contains monitors (1-*).
rule MainWithMonitors {
   from
      i : Program!Program (
         thisModule.monitorsDefined
      )
   to
      prg : Program!Program (
         name <- i.name,
         variables <- i.getVariables(),
         procedures <- i.getProcedures(),
         monitors <- Set{}
      ),

      -- 'this' variable delcaration: this variable is used for generated
      -- condition procedure calls.
      this_var : Program!VariableDeclaration (
         name <- 'this',
         type <- this_type
      ),
      -- 'this' type is 'program'
      this_type : Program!Type (
         name <- 'program'
      ),

      -- Basic types required for generated semaphore-based
      -- procedures.
      boolean_type : Program!Type (
         name <- 'boolean'
      ),
      integer_type : Program!Type (
         name <- 'integer'
      )
}

-- Rule 'MainWithoutMonitors'.
-- This rule generates the structure of the root Program element when the
-- input element contains no monitors.
```

```
rule MainWithoutMonitors {
   from
      i : Program!Program (
         not thisModule.monitorsDefined
      )
   to
      prg : Program!Program (
         name <- i.name,
         variables <- i.variables,
         procedures <- i.procedures,
         monitors <- Set{}
      )
}

-- Rule 'Monitor'.
-- This rule generates a set of variable and procedure declarations that are
-- associated with each defined monitor in the input model:
--   * the 'urgent' variable initialized to 'false';
--   * the 'urgentcount' variable initialized to '0';
--   * the 'mutex' variable initialized to 'true'.
rule Monitor {
   from
      i : Program!Monitor
   to
      -- Generation of a semaphore ('mutex') for mutual exclusion:
      --    * its name is prefixed by the monitor name;
      --    * its type is 'boolean';
      --    * its initial value is 'true'.
      mutex : Program!VariableDeclaration (
         name <- i.name + '_mutex',
         type <- thisModule.resolveTemp(thisModule.rootElt, 'boolean_type'),
         initialValue <- true_value,
         structure <- i.program
      ),

      -- Generation of a second semaphore ('urgent'):
      --    * its name is prefixed by the monitor name;
      --    * its type is 'boolean';
      --    * its initial value is 'false'.
      urgent : Program!VariableDeclaration (
         name <- i.name + '_urgent',
         type <- thisModule.resolveTemp(thisModule.rootElt, 'boolean_type'),
         initialValue <- false_value,
         structure <- i.program
      ),

      -- Generation of an integer counter ('urgentcount'):
      --    * its name is prefixed by the monitor name;
      --    * its type is 'integer';
      --    * its initial value is '0'.
      urgentcount : Program!VariableDeclaration (
         name <- i.name + '_urgentcount',
         type <- thisModule.resolveTemp(thisModule.rootElt, 'integer_type'),
         initialValue <- zero_value,
         structure <- i.program
      ),

      -- Basic values elements required for generated variables
      -- initialization.
      true_value : Program!BooleanExp (
         symbol <- true
      ),
      false_value : Program!BooleanExp (
         symbol <- false
      ),
      zero_value : Program!IntegerExp (
         symbol <- 0
      )
}

-- Rule 'Condition'.
-- This rule generates a set of variable and procedure declarations that are
-- associated with each defined 'condition' variable of the input model:
--   * the 'sem' variable initialized to 'false';
--   * the 'count' variable initialized to '0';
--   * the condition 'wait' procedure declaration;
--   * the condition 'signal' procedure declaration.
rule Condition {
   from
      condition : Program!VariableDeclaration (
         thisModule.allCondVariables->includes(condition)
      )
   to
      -- Generation of the 'sem' boolean variable:
      --    * its name is prefixed both by the monitor and the condition
      ---     names;
      --    * its type is 'boolean';
      --    * its initial value is 'false'.
      condsem : Program!VariableDeclaration (
         name <- condition.structure.name + '_' + condition.name +'_sem',
         type <- thisModule.resolveTemp(thisModule.rootElt, 'boolean_type'),
         initialValue <- false_value,
         structure <- condition.structure.program
      ),

      -- Generation of the 'count' integer variable:
      --    * its name is prefixed both by the monitor and the condition
      ---     names;
      --    * its type is 'integer';
      --    * its initial value is '0'.
      count : Program!VariableDeclaration (
         name <- condition.structure.name + '_' + condition.name + '_count',
         type <- thisModule.resolveTemp(thisModule.rootElt, 'integer_type'),
         initialValue <- zero_value_1,
         structure <- condition.structure.program
      ),


      -------------------------------------------------------------------
      -------------------------------------------------------------------
      -- Generation of the 'wait' procedure associated with a condition.
      -- The procedure name is built by associating: the monitor name, the
      -- condition name, and the 'wait' constant.
      -- It is composed of four statements:
      --    [1] condcount := condcount + 1;
      --    [2] if urgentcount > 0 then V(urgent) else V(mutex);
      --    [3] P(condsem);
      --    [4] condcount := condcount -1;
      cond_wait : Program!Procedure (
         name <- condition.structure.name + '_' + condition.name + '_wait',
         statements <-
            Sequence{wait_stat1, wait_stat2, wait_stat3, wait_stat4},
         container <- condition.structure.program
      ),
      wait_stat1 : Program!AssignmentStat (
         target <- count_exp_1,
         value <- plus
      ),
      wait_stat2 : Program!ConditionalStat (
         condition <- urgentcount_test,
         thenStats <- Sequence{then_st},
         elseStats <- Sequence{else_st}
      ),
      wait_stat3 : Program!ExpressionStat (
         expression <- third_exp
      ),
      wait_stat4 : Program!AssignmentStat (
         target <- count_exp_3,
         value <- less
      ),

      -- [Wait_1] condcount := condcount + 1;
      count_exp_1 : Program!VariableExp (
         declaration <- count
      ),
      plus : Program!OperatorCallExp (
         name <- '+',
         source <- count_exp_2,
         right <- one_value_1
      ),
      count_exp_2 : Program!VariableExp (
         declaration <- count
      ),
      -- [Wait_2] if urgentcount > 0 then V(urgent) else V(mutex);
      urgentcount_test : Program!OperatorCallExp (
         name <- '>',
         source <- urgentcount_exp_3,
         right <- zero_value_2
      ),
      urgentcount_exp_3 : Program!VariableExp (
         declaration <-
            thisModule.resolveTemp(condition.structure, 'urgentcount')
      ),
      then_st : Program!ExpressionStat (
         expression <- then_exp
      ),
      then_exp : Program!ProcedureCallExp (
         name <- 'v',
         source <-  urgent_exp_1,
         arguments <- Sequence{}
      ),
      urgent_exp_1 : Program!VariableExp (
         declaration <-
            thisModule.resolveTemp(condition.structure, 'urgent')
      ),
      else_st : Program!ExpressionStat (
         expression <- else_exp
      ),
      else_exp : Program!ProcedureCallExp (
         name <- 'v',
         source <- mutex_exp,
         arguments <- Sequence{}
      ),
      mutex_exp : Program!VariableExp (
         declaration <- thisModule.resolveTemp(condition.structure, 'mutex')
      ),
      -- [Wait_3] P(condsem);
      third_exp : Program!ProcedureCallExp (
         name <- 'p',
         source <- condsem_exp_1,
         arguments <- Sequence{}
      ),
      condsem_exp_1 : Program!VariableExp (
         declaration <- condsem
      ),
      -- [Wait_4] condcount := condcount -1;
      count_exp_3 : Program!VariableExp (
         declaration <- count
      ),
      less : Program!OperatorCallExp (
         name <- '-',
         source <- count_exp_4,
         right <- one_value_2
      ),
      count_exp_4 : Program!VariableExp (
         declaration <- count
      ),


      -----------------------------------------------------------------------
      -----------------------------------------------------------------------
      -- Generation of the 'signal' procedure associated with a condition.
      -- The procedure name is built by associating: the monitor name, the
      -- condition name, and the 'signal' constant.
      -- It is composed of three statements:
      --   [1] urgentcount := urgentcount +1;
      --   [2] if condcount > 0 then {V(condsem); P(urgent)};
      --   [3] urgentcount := urgentcount -1
      cond_signal : Program!Procedure (
         name <-
            condition.structure.name + '_' + condition.name + '_signal',
         statements <- Sequence{signal_stat1, signal_stat2, signal_stat3},
         container <- condition.structure.program
      ),
      signal_stat1 : Program!AssignmentStat (
         target <-urgentcount_exp_1,
         value <- urgentcount_plus
      ),
      signal_stat2 : Program!ConditionalStat (
         condition <- condcount_test,
         thenStats <- Sequence{signal_condsem_stat, wait_urgent_stat}
      ),
      signal_stat3 : Program!AssignmentStat (
         target <-urgentcount_exp_2,
         value <- urgentcount_less
      ),

      -- [Signal_1] urgentcount := urgentcount +1;
      urgentcount_exp_1 : Program!VariableExp (
         declaration <-
            thisModule.resolveTemp(condition.structure, 'urgentcount')
      ),
      urgentcount_plus : Program!OperatorCallExp (
         name <- '+',
         source <- urgentcount_exp_4,
         right <- one_value_3
      ),
      urgentcount_exp_4 : Program!VariableExp (
         declaration <-
            thisModule.resolveTemp(condition.structure, 'urgentcount')
      ),
      -- [Signal_2] if condcount > 0 then {V(condsem); P(urgent)};
      condcount_test : Program!OperatorCallExp (
         name <- '>',
         source <- count_exp_5,
         right <- zero_value_3
      ),
      count_exp_5 : Program!VariableExp (
         declaration <- count
      ),
      signal_condsem_stat : Program!ExpressionStat (
         expression <- signal_condsem
      ),
      signal_condsem : Program!ProcedureCallExp (
         name <- 'v',
         source <- condsem_exp_2,
         arguments <- Sequence{}
      ),
      condsem_exp_2 : Program!VariableExp (
         declaration <- condsem
      ),
      wait_urgent_stat : Program!ExpressionStat (
         expression <- wait_urgent
      ),
      wait_urgent : Program!ProcedureCallExp (
         name <- 'p',
         source <- urgent_exp_2,
         arguments <- Sequence{}
      ),
      urgent_exp_2 : Program!VariableExp (
         declaration <-
            thisModule.resolveTemp(condition.structure, 'urgent')
      ),
      -- [Signal_3] urgentcount := urgentcount -1
      urgentcount_exp_2 : Program!VariableExp (
         declaration <-
            thisModule.resolveTemp(condition.structure, 'urgentcount')
      ),
      urgentcount_less : Program!OperatorCallExp (
         name <- '-',
         source <- urgentcount_exp_5,
         right <- one_value_4
      ),
      urgentcount_exp_5 : Program!VariableExp (
         declaration <-
            thisModule.resolveTemp(condition.structure, 'urgentcount')
      ),


      -- Basic values elements required for generated variables
      -- initialization.
      false_value : Program!BooleanExp (
         symbol <- false
      ),
      one_value_1 : Program!IntegerExp (
         symbol <- 1
      ),
      one_value_2 : Program!IntegerExp (
         symbol <- 1
      ),
      one_value_3 : Program!IntegerExp (
         symbol <- 1
      ),
      one_value_4 : Program!IntegerExp (
         symbol <- 1
      ),
      zero_value_1 : Program!IntegerExp (
         symbol <- 0
      ),
      zero_value_2 : Program!IntegerExp (
         symbol <- 0
      ),
      zero_value_3 : Program!IntegerExp (
         symbol <- 0
      )
}

-- Rule 'StandaloneProcedure'.
-- This rule copies each standalone procedure defined in the input model to the
-- output model.
rule StandaloneProcedure {
   from
      i : Program!Procedure (
         i.container.oclIsKindOf(Program!Program)
      )
   to
      o : Program!Procedure (
         name <- i.name,
         parameters <- i.parameters,
         variables <- i.variables,
         statements <- i.statements,
         container <- i.container
      )
}

-- Rule 'MonitorProcedure'.
-- This rule copies each procedure defined in an input model monitor to the
-- output model.
-- The name of the generated procedure is prefixed by the monitor name.
-- The input procedure statements are preceded by a monitor entering statement
-- and followed by a monitor leaving statement.
rule MonitorProcedure {
   from
      i : Program!Procedure (
         i.container.oclIsKindOf(Program!Monitor)
      )
   to
      -- Generated procedure
      proc : Program!Procedure (
         name <- i.container.name + '_' + i.name,
         parameters <- i.parameters,
         variables <- i.variables,
         statements <- Sequence{in_stat, i.statements, out_stat},
         container <- i.container.program
      ),

      -- Monitor entering statement: P(mutex)
      in_stat : Program!ExpressionStat (
         expression <- in_exp
      ),
      in_exp : Program!ProcedureCallExp (
         name <- 'p',
         source <- mutex_exp1,
         arguments <- Sequence{}
      ),
      mutex_exp1 : Program!VariableExp (
         declaration <- thisModule.resolveTemp(i.container, 'mutex')
      ),

      -- Monitor leaving statement:
      -- if urgentcount > 0 then V(urgent); else V(mutex);
      out_stat : Program!ConditionalStat (
         condition <- urgentcount_test,
         thenStats <- Sequence{then_st},
         elseStats <- Sequence{else_st}
      ),
      -- Condition
      urgentcount_test : Program!OperatorCallExp (
         name <- '>',
         source <- count_exp,
         right <- zero_value
      ),
      count_exp : Program!VariableExp (
         declaration <- thisModule.resolveTemp(i.container, 'urgentcount')
      ),
      zero_value : Program!IntegerExp (
         symbol <- 0
      ),
      -- 'then' statement
      then_st : Program!ExpressionStat (
         expression <- then_exp
      ),
      then_exp : Program!ProcedureCallExp (
         name <- 'v',
         source <- urgent_exp,
         arguments <- Sequence{}
      ),
      urgent_exp : Program!VariableExp (
         declaration <- thisModule.resolveTemp(i.container, 'urgent')
      ),
      -- 'else' statement
      else_st : Program!ExpressionStat (
         expression <- else_exp
      ),
      else_exp : Program!ProcedureCallExp (
         name <- 'v',
         source <- mutex_exp2,
         arguments <- Sequence{}
      ),
      mutex_exp2 : Program!VariableExp (
         declaration <- thisModule.resolveTemp(i.container, 'mutex')
      )
}

-- Rule 'Type'.
-- This rule copies some of the types defined in the input model into the
-- output model. Each matched type must:
--   *  not be a 'condition' type, if no monitors are defined;
--   *  not be a 'condition', an 'integer' or a 'boolean' type, if some
--      monitors are defined (in this case, 'integer' and 'boolean' types
--      are generated once into the 'MainWithMonitors' rule).
rule Type {
   from
      i : Program!Type (
         i.name <> 'condition' and
         (
            (not thisModule.monitorsDefined) or
            (
               thisModule.monitorsDefined and
               i.name <> 'boolean' and
               i.name <> 'integer'
            )
         )
      )
   to
      o : Program!Type (
         name <- i.name
      )
}

-- Rule 'BooleanIntegerMonitorVariableDeclaration'.
-- This rule performs a copy of variable declarations of type 'integer' or
-- 'boolean 'that are declared either within a monitor or a procedure of a
-- monitor.
rule BooleanIntegerMonitorVariableDeclaration {
   from
      i : Program!VariableDeclaration (
         i.oclIsTypeOf(Program!VariableDeclaration) and
         i.isDeclaredInMonitor() and
         (
            i.type.name = 'integer' or
            i.type.name = 'boolean'
         )
      )
   to
      o : Program!VariableDeclaration (
         name <- i.structure.name + '_' + i.name,
         type <- i.getType(),
         initialValue <- i.initialValue,
         structure <- i.getTargetStructure()
      )
}

-- Rule 'MonitorVariableDeclaration'.
-- This rule performs a copy of each variable declaration which type is different
of
-- 'integer', 'boolean' and 'condition', and that is declared either within a
monitor
-- or a procedure of a monitor.
rule MonitorVariableDeclaration {
   from
      i : Program!VariableDeclaration (
         i.oclIsTypeOf(Program!VariableDeclaration) and
         i.isDeclaredInMonitor() and
         i.type.name <> 'condition' and
         i.type.name <> 'integer' and
         i.type.name <> 'boolean'
      )
   to
      o : Program!VariableDeclaration (
         name <- i.structure.name + '_' + i.name,
         type <- i.type,
         initialValue <- i.initialValue,
         structure <- i.getTargetStructure()
      )
}
```

```
-- Rule 'BooleanIntegerVariableDeclaration'.
-- This rule performs a copy of variable declarations of type 'integer' or
-- 'boolean' that are declared either within the root program or a standalone
-- procedure.
rule BooleanIntegerStandaloneVariableDeclaration {
   from
      i : Program!VariableDeclaration (
         i.oclIsTypeOf(Program!VariableDeclaration) and
         not i.isDeclaredInMonitor() and
         (
            i.type.name = 'integer'
            or
            i.type.name = 'boolean'
         )
      )
   to
      o : Program!VariableDeclaration (
         name <- i.name,
         type <- i.getType(),
         initialValue <- i.initialValue,
         structure <- i.structure
      )
}

-- Rule 'StandaloneVariableDeclaration'.
-- This rule performs of each variable declaration which type is different of
-- 'integer', 'boolean' and 'condition', and that is declared either within the
-- root program or a standalone procedure.
rule StandaloneVariableDeclaration {
   from
      i : Program!VariableDeclaration (
         i.oclIsTypeOf(Program!VariableDeclaration) and
         not i.isDeclaredInMonitor() and
         i.type.name <> 'condition' and
         i.type.name <> 'integer' and
         i.type.name <> 'boolean'
      )
   to
      o : Program!VariableDeclaration (
         name <- i.name,
         type <- i.type,
         initialValue <- i.initialValue,
         structure <- i.structure
      )
}

-- Rule 'BooleanIntegerMonitorParameter'.
-- If monitors are defined in the input program, this rule performs a copy,
-- from the input model to the output model, of each parameter of type
-- 'integer' or 'boolean'.
rule BooleanIntegerMonitorParameter {
   from
      i : Program!Parameter (
         thisModule.monitorsDefined and
         (
            i.type.name = 'integer' or
            i.type.name = 'boolean'
         )
      )
   to
      o : Program!Parameter (
            name <- i.name,
            type <- i.getType(),
            initialValue <- i.initialValue,
            direction <- i.direction,
            procedure <- i.procedure
        )
}

-- Rule 'Parameter'.
-- This rule performs a copy, from the input model to the output model, of a
-- parameter when:
--   * no monitors are defined in the input program;
--   * or monitors are defined in the input program but the parameter is not
--     of type 'integer' or 'boolean'.
rule Parameter {
    from
        i : Program!Parameter (
            not thisModule.monitorsDefined or
            (
                thisModule.monitorsDefined and
                i.type.name <> 'integer' and
                i.type.name <> 'boolean'
             )
        )
    to
        o : Program!Parameter (
            name <- i.name,
            type <- i.type,
            initialValue <- i.initialValue,
            direction <- i.direction,
            procedure <- i.procedure
        )
}

-------------
-- Statements
-------------
-- Rule 'ExpressionStat'.
-- This rule copies each expression statement from the input model to the
-- output model.
rule ExpressionStat {
    from
        i : Program!ExpressionStat
    to
        o : Program!ExpressionStat (
            expression <- i.expression
        )
}

-- Rule 'WhileStat'.
-- This rule copies each while statement from the input model to the
-- output model.
rule WhileStat {
    from
        i : Program!WhileStat
    to
        o : Program!WhileStat (
            condition <- i.condition,
            doStats <- i.doStats
        )
}
```

```
-- Rule 'ConditionalStat'.
-- This rule copies each conditional statement from the input model to the
-- output model.
rule ConditionalStat {
   from
      i : Program!ConditionalStat
   to
      o : Program!ConditionalStat (
         condition <- i.condition,
         thenStats <- i.thenStats,
         elseStats <- i.elseStats
      )
}

-- Rule 'AssignmentStat'.
-- This rule copies each assignment statement from the input model to the
-- output model.
rule AssignmentStat {
   from
      i : Program!AssignmentStat
   to
      o : Program!AssignmentStat (
         target <- i.target,
         value <- i.value
      )
}

--------------
-- Expressions
--------------
-- Rule 'OperatorCallExp'.
-- This rule copies each operator call expression from the input model to the
-- output model.
rule OperatorCallExp {
   from
      i : Program!OperatorCallExp
   to
      o : Program!OperatorCallExp (
         right <- i.right,
         source <- i.source,
         name <- i.name
      )
}

-- Rule 'AttributeCallExp'.
-- This rule copies each attribute call expression from the input model to the
-- output model.
rule AttributeCallExp {
   from
      i : Program!AttributeCallExp
   to
      o : Program!AttributeCallExp (
         source <- i.source,
         name <- i.name
      )
}

-- Rule 'ProcedureCallExp'.
-- This rule copies each procedure call expression, whose source is different
-- from 'condition', from the input model to the output model.
rule ProcedureCallExp {
   from
```

```
      i : Program!ProcedureCallExp (
         i.source.declaration.type.name <> 'condition'
      )
   to
      o : Program!ProcedureCallExp (
         arguments <- i.arguments,
         source <- i.source,
         name <- i.name
      )
}

-- Rule 'ConditionProcCallExp'.
-- This rule copies each 'condition' call expression from the input model to
-- the output model.
-- In the output model, the source call is set to the global 'this' variable
-- defined in the 'MainWithMonitors' rule.
rule ConditionProcCallExp {
   from
      i : Program!ProcedureCallExp (
         i.source.declaration.type.name = 'condition'
      )
   to
      call_exp : Program!ProcedureCallExp (
         arguments <- i.arguments,
         source <- source_exp,
         name <- i.source.declaration.structure.name + '_'
            + i.source.declaration.name + '_' + i.name
      ),
      source_exp : Program!VariableExp (
         declaration <-
            thisModule.resolveTemp(thisModule.rootElt, 'this_var')
      )
}

-- Rule 'VariableExp'.
-- This rule copies each variable expression, which is not a 'condition'
-- variable, from the input model to the output model.
rule VariableExp {
   from
      i : Program!VariableExp (
         thisModule.allCondCalls->excludes(i)
      )
   to
      o : Program!VariableExp (
         declaration <- i.declaration
      )
}

-- Rule 'BooleanExp'.
-- This rule copies each boolean value from the input model to the output
-- model.
rule BooleanExp {
   from
      i : Program!BooleanExp
   to
      o : Program!BooleanExp (
         symbol <- i.symbol
      )
}

-- Rule 'IntegerExp'.
-- This rule copies each integer value from the input model to the output
-- model.
rule IntegerExp {
   from
      i : Program!IntegerExp
   to
      o : Program!IntegerExp (
         symbol <- i.symbol
      )
}
```

I. Program metamodel in KM3 format

```
package Program {

   abstract class LocatedElement {
      attribute location : String;
   }

   abstract class NamedElement extends LocatedElement {
      attribute name : String;
   }

   abstract class Structure extends NamedElement {
      reference variables[*] ordered container : VariableDeclaration oppositeOf
structure;
   }

   abstract class ProcContainerElement extends Structure {
      reference procedures[*] ordered container : Procedure oppositeOf "container";
   }

   class Program extends ProcContainerElement {
      reference monitors[*] ordered container : Monitor oppositeOf program;
   }

   class Monitor extends ProcContainerElement {
      reference program : Program oppositeOf monitors;
   }

   -- Procedures
   class Procedure extends Structure {
      reference "container" : ProcContainerElement oppositeOf procedures;
      reference parameters[*] ordered container : Parameter oppositeOf procedure;
      reference statements[*] ordered container : Statement;
   }

   class VariableDeclaration extends NamedElement {
      reference type : Type;
      reference initialValue[0-1] container : Expression;
      reference structure : Structure oppositeOf variables;
   }

   class Parameter extends VariableDeclaration {
      attribute direction : Direction;
      reference procedure : Procedure oppositeOf parameters;
   }

   enumeration Direction {
      literal in;
      literal out;
   }
   -- End Procedures

   -- Types
   class Type extends NamedElement {
   }
   -- End Types

   -- Expressions
   abstract class Expression extends LocatedElement {
   }
```

```
    class VariableExp extends Expression {
        reference declaration : VariableDeclaration;
    }

    -- PropertyCalls
    abstract class PropertyCallExp extends Expression {
        reference source container : Expression;
        attribute name : String;
    }

    class OperatorCallExp extends PropertyCallExp {
        reference right container : Expression;
    }

    class AttributeCallExp extends PropertyCallExp {
    }

    class ProcedureCallExp extends PropertyCallExp {
        reference arguments[*] ordered container : Expression;
    }
    -- End PropertyCalls

    -- Literals
    abstract class LiteralExp extends Expression {
    }

    class BooleanExp extends LiteralExp {
        attribute symbol : Boolean;
    }

    class IntegerExp extends LiteralExp {
        attribute symbol : Integer;
    }
    -- End Literals
    -- End Expressions

    -- Statements
    abstract class Statement extends LocatedElement {
    }

    class AssignmentStat extends Statement {
        reference target container : VariableExp;
        reference value container : Expression;
    }

    class ConditionalStat extends Statement {
        reference condition container : Expression;
        reference thenStats[1-*] container : Statement;
        reference elseStats[*] container : Statement;
    }

    class WhileStat extends Statement {
        reference condition container : Expression;
        reference doStats[1-*] container : Statement;
    }

    class ExpressionStat extends Statement {
        reference expression container : Expression;
    }
    -- End Statements
}
```

References

[1] Hoare C.A.R. Monitors: An Operating System Structuring Concept. In *Communications of the ACM*, Volume 17, Number 10, October 1974.

[2] Dijkstra E. W. Cooperating Sequential Processes. In *Programming Languages* (Ed. F. Genuys). Academic Press, New York, 1968.

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