Solving the TTC 2011 Compiler Optimization Case with QVTR-XSLT

Dan Li [*], Xiaoshan Li
Faculty of Science and Technology, University of Macau, China
lidan@iist.unu.edu, xsl@umac.mo

Volker Stolz
Department of Informatics, University of Oslo, Norway & UNU-IIST, Macau, China
stolz@ifi.uio.no

1 Introduction

In this short paper we present our solution for the Compiler Optimization case study [1] of the Transformation Tool Contest (TTC) 2011 using the QVTR-XSLT tool [2]. The tool supports editing and execution of the graphical notation of QVT Relations language [6].

The case study addresses the problem of optimizing the *intermediate representation* of compiled program code. This problem consists of two tasks: local optimization and instruction selection. The first task mainly concerns *constant folding* which evaluates operations with only constant operands, corresponding control flows are also optimized. The instruction selection task transforms the intermediate representation into a target representation of similar structure. The SHARE demo related to the paper can be found at [3].

We begin by giving a brief introduction of the QVTR-XSLT tool in Section 2. Section 3 explains the design of transformations for the case study. We discuss the experimental result and evaluation of the solution against the criteria given in the case specification in Section 4.

2 The QVTR-XSLT tool

Model transformation is the core technology for model-driven development, and is used in software model refinement, evolution, refactoring and code generation. To address the need for a common transformation language, the Object Management Group (OMG) proposed the Query/View/Transformation language (QVT) [6] standard. QVT has a hybrid declarative/imperative nature. In its declarative language, called QVT Relations (QVTR), a transformation is defined as a set of *relations* (rules) between source and target models, each conforming to their respective metamodels. Transformations are driven by a single, designated top-level relation.

QVTR combines a textual and a graphical notation. In graphical syntax, a relation specifies how two object diagrams, called *domain patterns*, relate to each other. That is, the *structural* matching of elements in the source- and target model is done diagrammatically. Moreover, QVTR employs a textual language based on essential OCL [5] to define additional (non-structural) constrains in relations. The

[*]On leave from Guizhou Academy of Sciences, Guizhou, China

graphical notation of QVTR provides a concise, intuitive and yet powerful way to specify transformations. However, currently there are very few tools supporting QVTR, and even fewer for its graphical notation.

QVTR-XSLT supports the graphical notation of QVT Relations, and an execution engine for a subset of QVTR by means of XSLT programs. It consists of two parts:

- **Graphical Editor**: Building on top of *MagicDraw UML* [4], the editor has a graphical interface for defining metamodels as simple class diagrams, specifying QVTR relations and queries in graphical notation, validating the design, and saving the transformations as an XML file.
- **Code generator**: It reads in the XML file, and generates an XSLT stylesheet for a transformation.

The outputs of the code generator are pure XSLT programs which can be directly executed under any XSLT processor on any platform. Additionally, we have also developed a transformation runner, in the form of a Java program invoking the Saxon 9 XSLT processor, to facilitate the execution of generated XSLT stylesheets.

The QVTR-XSLT tool supports transformation parameters, transformation inheritance through rule overriding, and multiple input and output models. Furthermore, *in-place* transformations are defined as modifications (insert, remove, replace) of the existing model elements. QVTR-XSLT-based transformations are used in the rCOS Modeler for use case-driven development of component- and object systems.

3 Transformation Design

The metamodel. As the first step of the transformation design, we define a simple metamodel for the intermediate representation (IR) of the FIRM model, as shown in Fig. 1. In the metamodel, a FIRM model consists of *Graphs*, and the transformations only deal with graphs of type *Default Graph*. Within a graph, a *Node* represents an operation, and the type of the operation is decided by the *xlink:href* property of its *Type* element. The property may be *#Jmp*, *#Add*, *#Mul*, or *#And*, etc. It also could be *#StartBlock*, *#Block*, or *#End* for a control flow node. A node may also own some *Attributes*, each of which has a name, and some values of different types. *Edges* specify the dependencies between nodes. An edge also has a type, such as *#Dataflow* or *#ControlFlow*, and a *position*.

We define a set of well-formedness rules as OCL class invariants for the metamodel:

- An instance of class *Attribute* has only two properties (one of them is the *name*).

context Attribute **inv** twoProperties: **self**.getAllProperties() $\rightarrow$ size()=2

where the *getAllProperties*() returns all properties of an instance.

- A *FirmMode* has at least one default graph.

context Graph **inv** hasDefault: Graph.allInstances()$\rightarrow$select(id='DefaultGraph')$\rightarrow$size()>=1

- Within a default graph, the *id* attribute is the unique identifier.

context Graph **inv** uniqueId:
  Graph.allInstances()$\rightarrow$select(id='DefaultGraph')$\rightarrow$ forAll(g|g.node.id$\rightarrow$asBag()
    $\rightarrow$union(g.edge.id$\rightarrow$asBag())$\rightarrow$isUnique(id))

Local optimizations. This task is a typical *in-place* transformation, in which both the input and output models are the same, and the execution of one rule could affect its subsequent rules. Since the execution unit of QVTR is a transformation, the optimization task is actually accomplished by a chain of executions

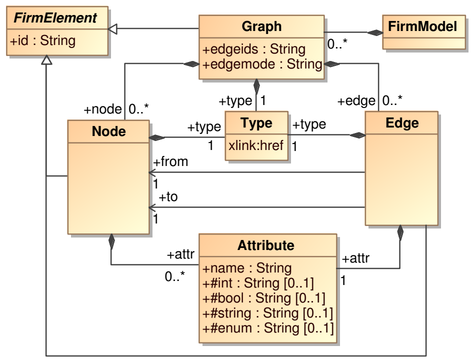


Figure 1: The metamodel for IR

of the transformation; each execution makes some changes to the model, and its output serves as the input of the following execution. Execution will stop if no more changes happen. This process is automated if running in the transformation runner.

The complete transformation consists of 13 relations, 9 queries and 2 functions (see Appendix B). Some of the relations are used for auxiliary purposes, such as removing *blocks*, *nodes* and *edges* from the model, or changing the *position* of an edge (Fig. 11–14). As in the model there is no direct navigation between nodes, or from a node to its connected edges, queries are defined to retrieve information such as the incoming and outgoing edges of a node, or a node's original and destination nodes. (Fig. 15–23). Some queries are functionally overlapping, as we want to use appropriate query names in different situations. All mathematical and logical calculations are performed by two functions. (Fig 24–25) . Because of the limited mathematical abilities of OCL expressions, we only deal with mathematical operations of *Add*, *Sub*, *Mul* and *Div*, and *LESS*, *EQUAL* and *GREATER* for logic operations.

The main part of the transformation definition has 9 relations that will be discussed in the following:

- **FirmModelTrans**: The transformation starts from this initial *top level* relation, which matches a graph with type of *DefaultGraph*, then the relation *FoldOper* and relation group *FoldNode* are invoked from the *where* clause.
- **FoldOper**: The relation first checks whether the node is a binary operation of the four mathematical kinds or a *Cmp*, then the query *GetToData* is called to get its two operands. If two of them are *Const*, the relation group *DoFoldOper* is invoked.
- **DoFoldOper**: This group includes two relations: *DoFoldCmp* and *DoFoldMath*. Inside the *DoFoldCmp* relation, values of the two const operands of the Cmp node are compared with the *CalcuLogic* function, and also compared with the incoming edges of the corresponding Cond node to decide which const operand should be removed. Then the Cond node and its connected edges are removed, and the Cmp node becomes a Jmp node. The *DoFoldMath* relation first calculates the mathematical result of the two consts, and then changes the operation node into a Const node and puts the result as the value of the node; finally the two outgoing edges of the operation node are removed.
- **FoldNode**: This group includes the following relations:
  - **FoldPhi** and **DoFoldPhi**: If there is only one outgoing dataflow edge for a Block where a Phi is located in, the *FoldPhi* relation invokes the *DoFoldPhi* relation, and the latter removes

the Phi node, relinks its users directly to the correct Const, and resets the *position* of the linking edge.

– **FoldJmpBlock**: The relation removes blocks which only contain a Jmp node.
– **FoldNoOutBlock**: Removes blocks without any outgoing control flow edges.
– **FoldIsolateConst**: Removes Const nodes which have no users.

Two XSLT stylesheet are generated for the transformation, with a total of 480 lines.

Instruction selection. The transformation for instruction selection is designed as a source to target model transformation, while both source and target models share the same metamodel. The complete transformation consists of 13 relations, 5 queries and 2 functions (see Appendix C). Many of the relations are used for trivial one-to-one copying from the source model to the target model (*CopyAtt, CopyNode, EdgeToEdge, OtherGraph*). The transformation starts from relation *FirmMode*, and then the relation groups *GraphToGraph* and *NodeToNode* are sequentially invoked, the latter includes relation *BinaryOp* and *UniqueOp*.

The major work of the transformation is accomplished by the following relations:

- **BinaryOp**: All binary operations are selected by the relation. For each operation *op*, we change its type *optp* to "*Target*"+*optp*, and invoke relation *MakeBinaryI*.
- **MakeBinaryI**: An additional new operation *top* is created with type of "*Target*"+ *optp*+"*I*", along with a new *value* attribute, and all connected edges of *op* are duplicated to *top* using relation *MakeEdge*. Moreover, if the commutative property of *op* is *false*, relation *Make-NewConst* is invoked.
- **MakeNewConst**: The relation creates a new Const node in the start block, and an outgoing edge with *position* 1 is also created to link *top* to the const node.
- **UniqueOp**: All other operations marked with "*" in the case specification are selected by the relation. The operation type *optp* is changed to "*Target*"+*optp*. If the operation is Load or Store, we invoke relation *MakeLoadStoreI*.
- **MakeLoadStoreI**: A new StoreI or LoadI operation node is created, which has an additional *symbol* attribute with a string value of "global".

Similar to the task of local optimizations, queries are used to retrieve information from the model. Function *GetTargetName* computes the target type from the type of an operation, and *GetNewId* generates a new identifier for a model element. An XSLT stylesheet of 330 lines of code is generated for the transformation.

In addition, we implement the model validating rules given in the case specification as an independent transformation. The generated XSLT stylesheet for the transformation is about 280 lines of code. It outputs a HTML page showing the results.

4 Experiments and Evaluation

Using our transformation runner, we execute the transformations on the examples provided by the case in a laptop of Intel M330 2.13 GHz CPU, 3 GB memory, and running Windows 7 home version. The DTD definition (second line) has to be removed from the .gxl file of each example to prevent the produce

Table 1: Result of the transformations for compiler optimization

Transformation	Example (.gxl)	Size (kb)	Time (ms)
Local Optimizations	min	36	155
	const	59	410
Instruction Selection	const	59	15
	mem	198	850
	testcase	186	820

of additional namespace information. The results are shown in Table 1. The execution time includes loading and saving model files from/to disk.

Our solution has covered all examples of the two tasks of the case study, except *zero.gxl*, which needs more math functions than our tool can provide, such as shifting and bit operations. As a high-level general purpose transformation languages, neither QVTR nor XSLT offer explicit parallelism, and leave this to a particular implementation. We are not aware that any XSLT processor makes use of parallelism except for an Intel research prototype.

The performance and the memory needed are much dependent on the XSLT processor used, and we can see from the results our tool works well, as it completes in under one second for every example. Our solution is *pure*, since no other code (e.g. hand-crafted XSLT) is required for the transformation of the examples, except for the iterative runner which applies the transformation until the result stabilises.

Conclusion Based on the QVTR-XSLT tool, we define a transformation using the graphical notation of QVT Relations, and generate an XSLT program to execute the transformation. Our contribution is two-fold: we have provided a solution for the two tasks of the compiler optimization case study of TTC 2011, and shown that our QVT-XSLT engine translates those examples, so that they can be executed in a standard XSLT engine.

Acknowledgements Partially supported by the ARV and GAVES grants of the Macau Science and Technology Development Fund, and the Guizhou International Scientific Cooperation Project G[2011]7023 and GY[2010]3033.

References

[1] Sebastian Buchwald & Edgar Jakumeit (2011): *Compiler Optimization: A Case for the Transformation Tool Contest*. In Pieter Van Gorp, Steffen Mazanek & Louis Rose, editors: *TTC 2011: Fifth Transformation Tool Contest, Zürich,Switzerland, June 29-30 2011*, EPTCS.

[2] Dan Li, Xiaoshan Li & Volker Stolz (2011): *QVT-based model transformation using XSLT*. *SIGSOFT Softw. Eng. Notes* 36, pp. 1–8, doi:`10.1145/1921532.1921563`.

[3] lidan@iist.unu.edu (2011): *Online demo: QVTR-XSLT solutions to the TTC11 Hello World and Compiler Optimization case studies*. Available at `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_TTC11_QVTR-XSLT.vdi`.

[4] NoMagic, Inc.: *MagicDraw*. `http://www.magicdraw.com`.

[5] Object Management Group (2006): *Object Constraint Language, version 2.0*. Available at `http://www.omg.org/spec/OCL/2.0/`.

[6] Object Management Group (2009): *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*.

A A Brief Introduction to QVT Relations

QVT Relations (QVTR) is a declarative model transformation language proposed by the OMG as part of the MOF Query/View/Transformations (QVT) standard [6]. QVTR specifies a *transformation* as a set of *relations* between source and target metamodels. A metamodel is defined in our tool as a simple class diagram. In addition, a transformation may own some *functions*, which are side-effect-free operations, and *queries* used to retrieve information from the models.

In the graphical notation, a *relation* defines how two object diagrams, called *domain patterns*, relate to each other. The object with tag ≪*domain*≫ is the *root* of a domain pattern, and it also serves as a parameter of the relation. In general, we assume the left domain pattern is the source domain, and the right the target domain. An *object* or a property of an object could be given a name that is taken as a *variable*. If the object is in the source domain pattern, then the object or the value of the property is bound to the variable. Otherwise the object in target domain pattern means assigning the value of the variable to the object or property. Note that a property variable in the diagrams may contain additional quote-characters that are an artefact of the visualization, and not string delimiters.

When a relation is executed, the source domain pattern is searched in the source model by way of *pattern matching* which starts from the domain root. When a match is found, all variables defined in source domain pattern are bound to values. The target domain pattern acts as a template to create corresponding objects and links in the target model using the values of the variables in the pattern.

A relation may define a pair of optional *when*- and *where*-clauses which consist of a set of OCL expressions. The *when*-clause indicates additional matching conditions for the relation. And new variables can be defined in the *where*-clause. Other relations could be invoked in the *where*-clause and variables can be passed as arguments. A relation may also have *primitive domains* in order to pass additional parameters between the relations. Furthermore, a relation is either designed as a *top-level* relation, or a *non-top-level* relation. A *top-level* relation is invoked from the transformation framework, and *non-top-level* relations are invoked by other relations.

B Transformation for Local Optimizations

• **Transformation configuration:** name : *TTC_LocalOptimizations*, isInPlace : *true*, rInPlace : *true*, source : *Intermediate*, sourceKey : *id*, sourceName : *original*, target: *Intermediate*, targetKey:*id*, targetName : *optim*.

B.1 QVTR relations

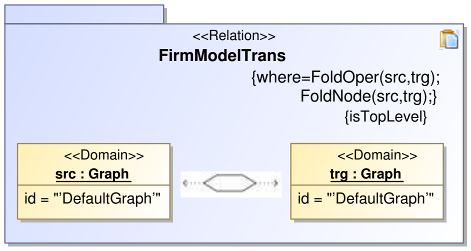


Figure 2: Starting top level relation

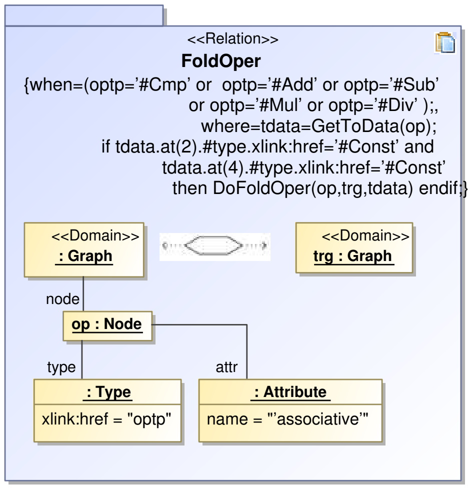

<<Relation>>
FoldOper
{when=(optp='#Cmp' or optp='#Add' or optp='#Sub'
or optp='#Mul' or optp='#Div');,
where=tdata=GetToData(op);
if tdata.at(2).#type.xlink:href='#Const' and
tdata.at(4).#type.xlink:href='#Const'
then DoFoldOper(op,trg,tdata) endif;}
<<Domain>>
: Graph
<<Domain>>
trg : Graph
node
op : Node
type
attr
: Type
xlink:href = "optp"
: Attribute
name = "'associative'"

Figure 3: Select binary operation with two const operands

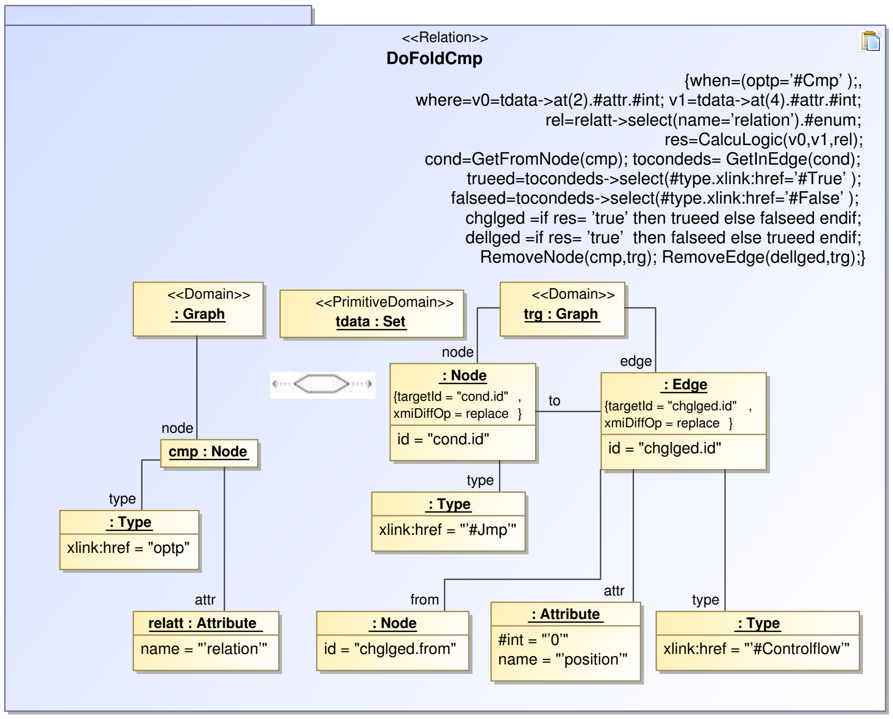

<<Relation>>
DoFoldCmp
{when=(optp='#Cmp');,
where=v0=tdata->at(2).#attr.#int; v1=tdata->at(4).#attr.#int;
rel=relatt->select(name='relation').#enum;
res=CalcuLogic(v0,v1,rel);
cond=GetFromNode(cmp); tocondeds= GetInEdge(cond);
trueed=tocondeds->select(#type.xlink:href='#True');
falseed=tocondeds->select(#type.xlink:href='#False');
chglged =if res= 'true' then trueed else falseed endif;
dellged =if res= 'true' then falseed else trueed endif;
RemoveNode(cmp,trg); RemoveEdge(dellged,trg);}
<<Domain>>
: Graph
<<PrimitiveDomain>>
tdata : Set
<<Domain>>
trg : Graph
node
edge
: Node
{targetId = "cond.id" ,
xmiDiffOp = replace }
id = "cond.id"
to
: Edge
{targetId = "chglged.id" ,
xmiDiffOp = replace }
id = "chglged.id"
node
cmp : Node
type
type
: Type
xlink:href = "optp"
: Type
xlink:href = "'#Jmp'"
attr
from
attr
type
relatt : Attribute
name = "'relation'"
: Node
id = "chglged.from"
: Attribute
#int = "'0'"
name = "'position'"
: Type
xlink:href = "'#Controlflow'"

Figure 4: Cope with Cmp operation (relName: *DoFoldOper*, rInPlace : *true*)

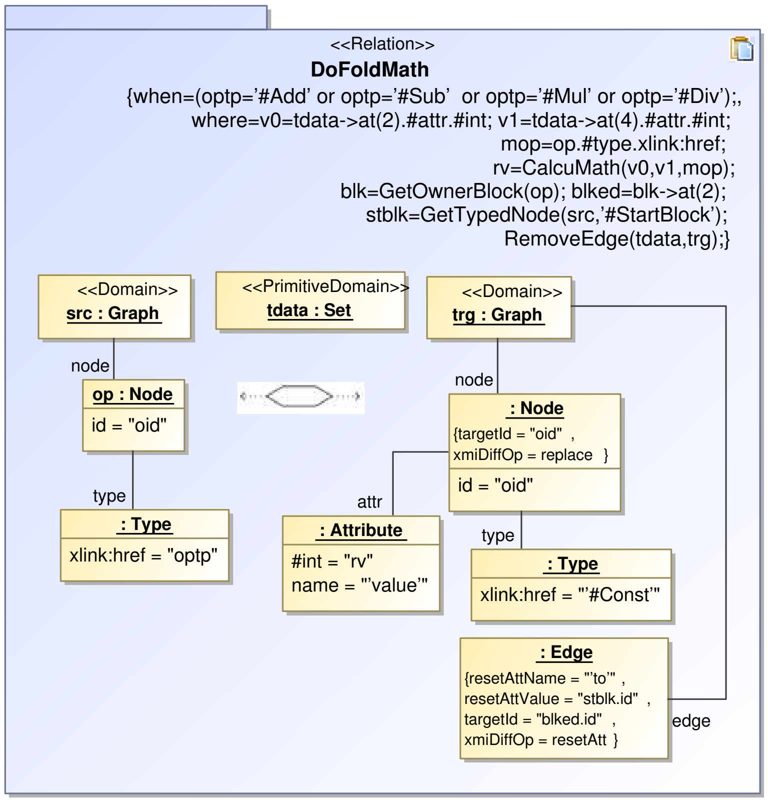

<<Relation>>
DoFoldMath
{when=(optp='#Add' or optp='#Sub' or optp='#Mul' or optp='#Div');,
where=v0=tdata->at(2).#attr.#int; v1=tdata->at(4).#attr.#int;
mop=op.#type.xlink:href;
rv=CalcuMath(v0,v1,mop);
blk=GetOwnerBlock(op); blked=blk->at(2);
stblk=GetTypedNode(src,'#StartBlock');
RemoveEdge(tdata,trg);}
<<Domain>> src : Graph
<<PrimitiveDomain>> tdata : Set
<<Domain>> trg : Graph
node
op : Node
id = "oid"
type
: Type
xlink:href = "optp"
node
: Node
{targetId = "oid" , xmiDiffOp = replace }
id = "oid"
attr
: Attribute
#int = "rv"
name = "'value'"
type
: Type
xlink:href = "'#Const'"
: Edge
{resetAttName = "'to'" , resetAttValue = "stblk.id" , targetId = "blked.id" , xmiDiffOp = resetAtt }
edge

Figure 5: Cope with math operations (relName: *DoFoldOper*, rInPlace : *true*)

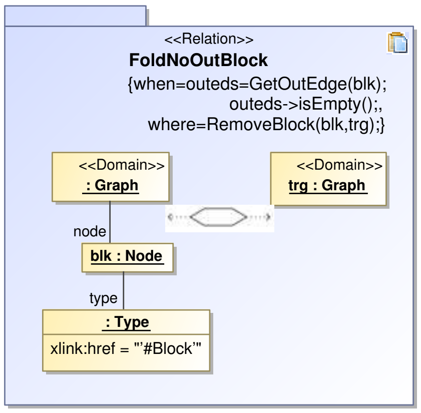


Figure 6: Fold block without outgoing edge (relName: *FoldNode*, rInPlace : *true*)

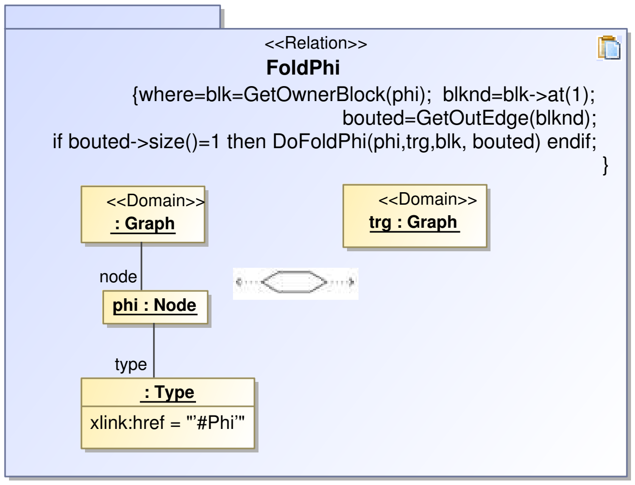
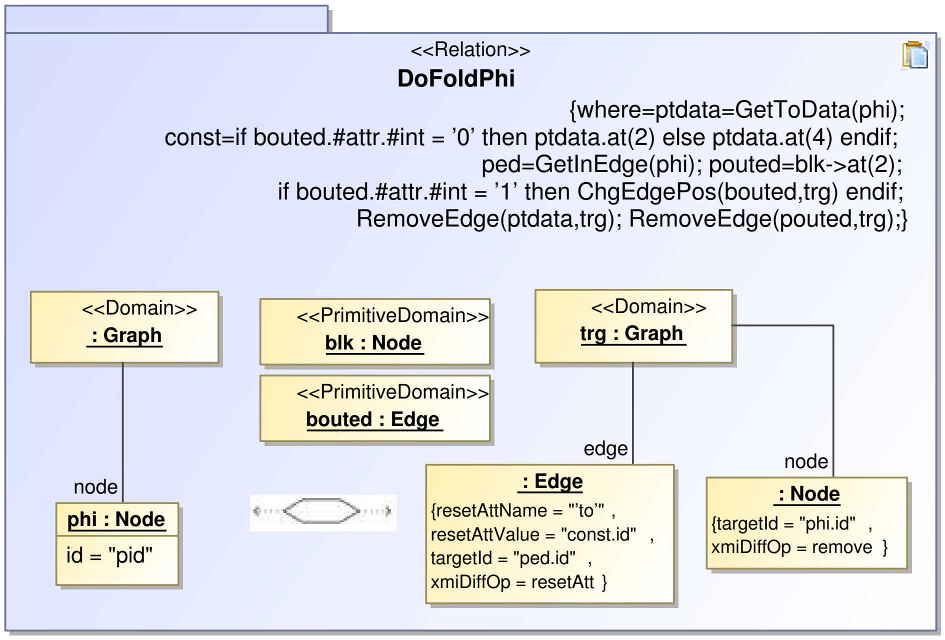
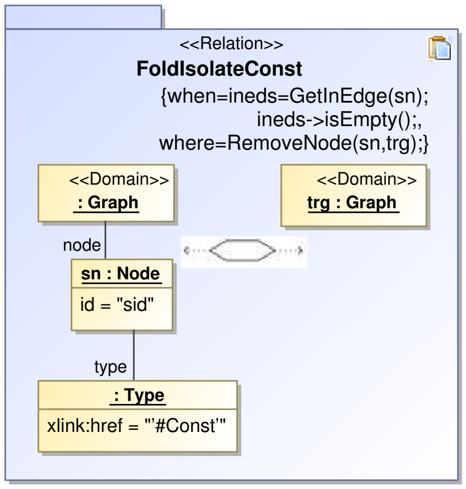

<<Relation>>
FoldPhi
{where=blk=GetOwnerBlock(phi); blknd=blk->at(1);
bouted=GetOutEdge(blknd);
if bouted->size()=1 then DoFoldPhi(phi,trg,blk, bouted) endif;
}
<<Domain>> : Graph
<<Domain>> trg : Graph
node
phi : Node
type
: Type
xlink:href = "'#Phi'"

Figure 7: Select a Phi owned by a block with only out control edge (relName: *FoldNode*)



<<Relation>>
DoFoldPhi
{where=ptdata=GetToData(phi);
const=if bouted.#attr.#int = '0' then ptdata.at(2) else ptdata.at(4) endif;
ped=GetInEdge(phi); pouted=blk->at(2);
if bouted.#attr.#int = '1' then ChgEdgePos(bouted,trg) endif;
RemoveEdge(ptdata,trg); RemoveEdge(pouted,trg);}
<<Domain>>
: Graph
<<PrimitiveDomain>>
blk : Node
<<PrimitiveDomain>>
bouted : Edge
<<Domain>>
trg : Graph
node
phi : Node
id = "pid"
edge
: Edge
{resetAttName = "'to'" ,
resetAttValue = "const.id" ,
targetId = "ped.id" ,
xmiDiffOp = resetAtt }
node
: Node
{targetId = "phi.id" ,
xmiDiffOp = remove }

Figure 8: Fold a Phi node (rInPlace : *true*)



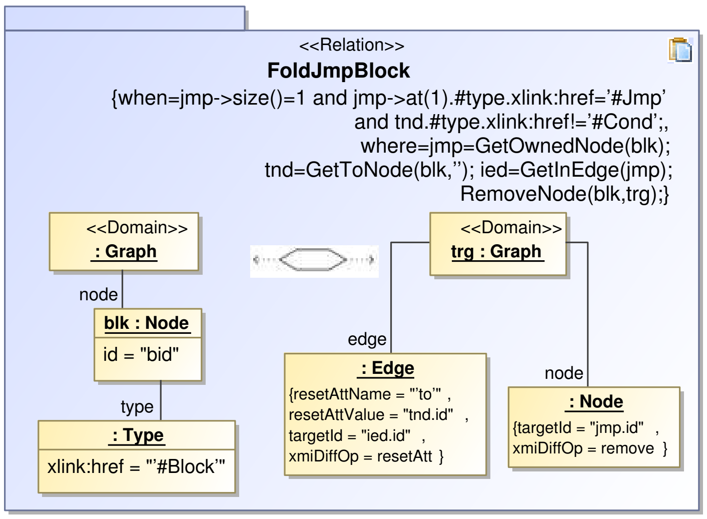


Figure 9: Fold a Const without incoming edges (relName: *FoldNode*, rInPlace : *true*)

Figure 10: Fold blocks containing only useless Jmp (relName: *FoldNode*, rInPlace : *true*)

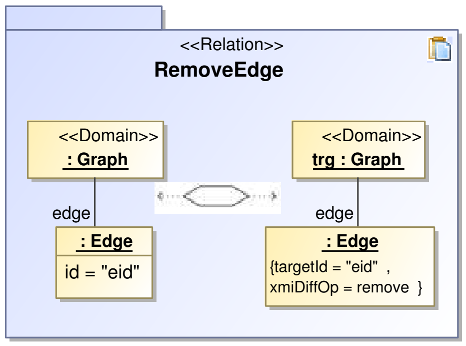


Figure 11: Remove an *Edge*

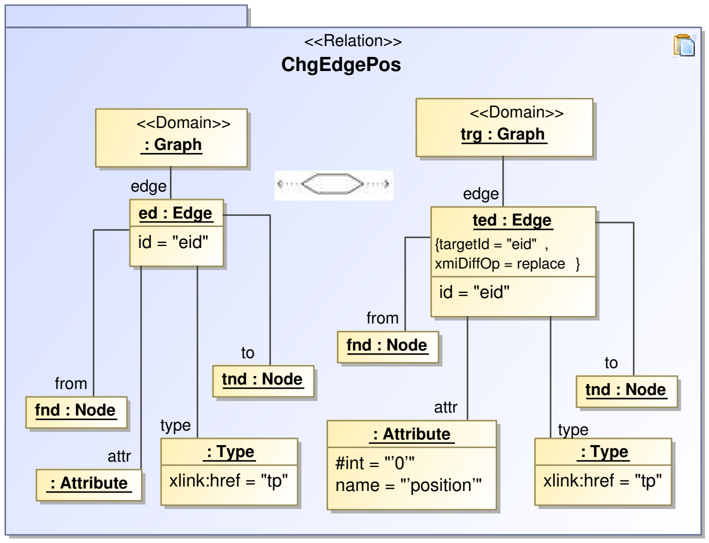
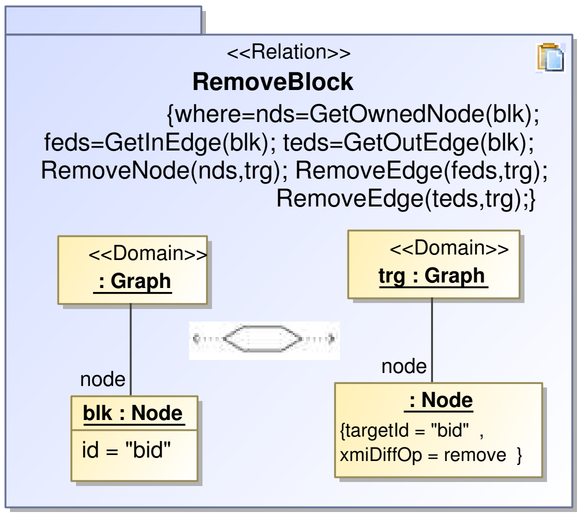


Figure 12: Change position of an *Edge*



<<Relation>>
RemoveBlock
{where=nds=GetOwnedNode(blk);
feds=GetInEdge(blk); teds=GetOutEdge(blk);
RemoveNode(nds,trg); RemoveEdge(feds,trg);
RemoveEdge(teds,trg);}
<<Domain>>
: Graph
<<Domain>>
trg : Graph
node
node
blk : Node
id = "bid"
: Node
{targetId = "bid" , xmiDiffOp = remove }

Figure 13: Remove a *Block*

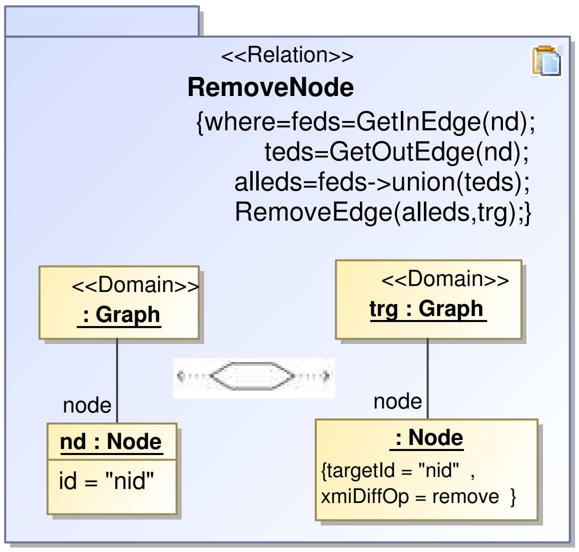


Figure 14: Remove a *Node*

B.2 Queries

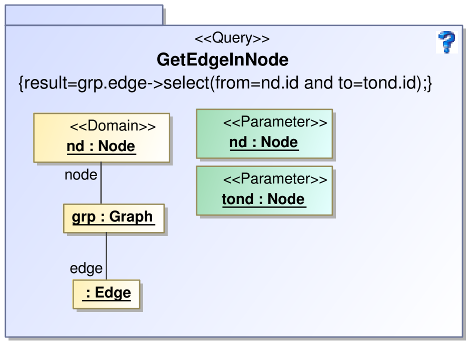


Figure 15: Get edges between nodes

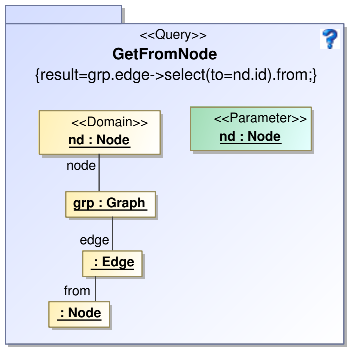


Figure 16: Get original nodes of a node

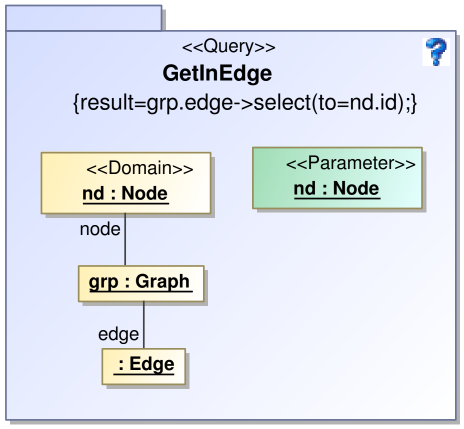


Figure 17: Get incoming edges of a node

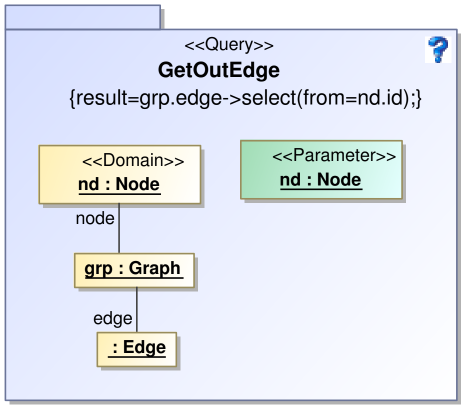


Figure 18: Get outgoing edges of a node

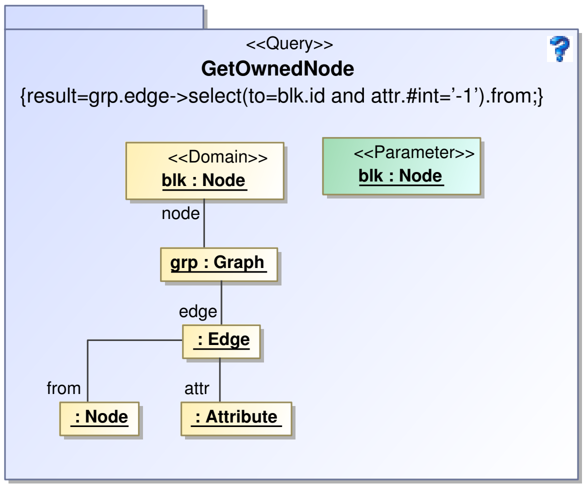


Figure 19: Get owned nodes of a block

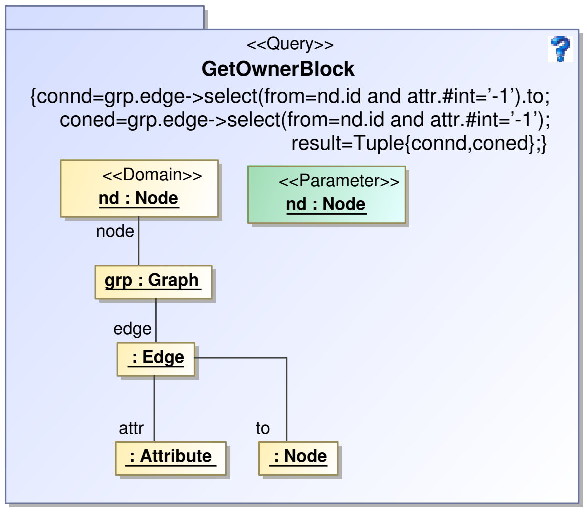

<<Query>>
GetOwnerBlock
{connd=grp.edge->select(from=nd.id and attr.#int='-1').to;
coned=grp.edge->select(from=nd.id and attr.#int='-1');
result=Tuple{connd,coned};}
<<Domain>>
nd : Node
<<Parameter>>
nd : Node
node
grp : Graph
edge
: Edge
attr
to
: Attribute
: Node

Figure 20: Get owner block of a node

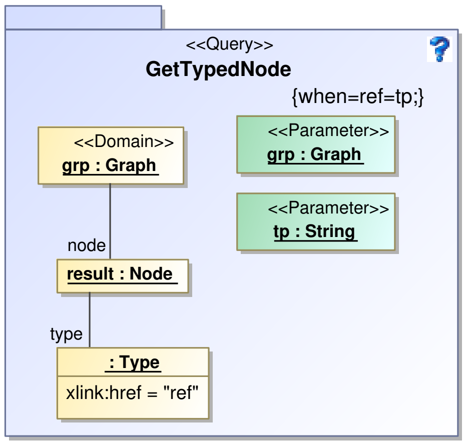


Figure 21: Get nodes of specific type

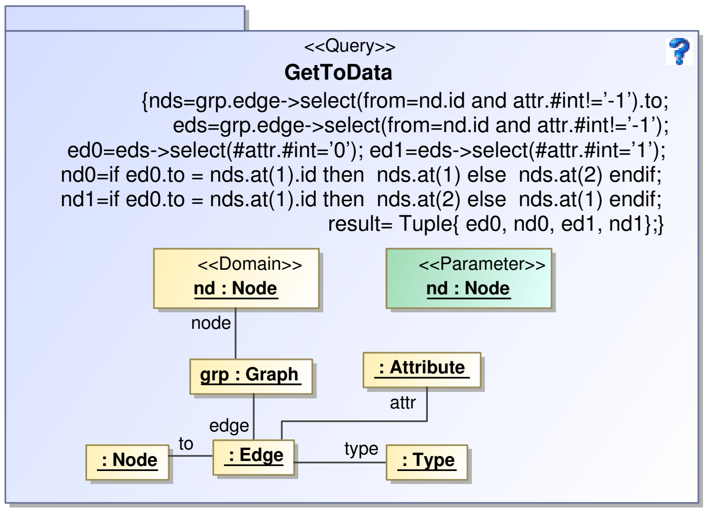

<<Query>>
GetToData
{nds=grp.edge->select(from=nd.id and attr.#int!='-1').to;
eds=grp.edge->select(from=nd.id and attr.#int!='-1');
ed0=eds->select(#attr.#int='0'); ed1=eds->select(#attr.#int='1');
nd0=if ed0.to = nds.at(1).id then nds.at(1) else nds.at(2) endif;
nd1=if ed0.to = nds.at(1).id then nds.at(2) else nds.at(1) endif;
result= Tuple{ ed0, nd0, ed1, nd1};}
<<Domain>>
nd : Node
<<Parameter>>
nd : Node
node
grp : Graph
: Attribute
attr
edge
: Node
to
: Edge
type
: Type

Figure 22: Get edges and nodes of data operands

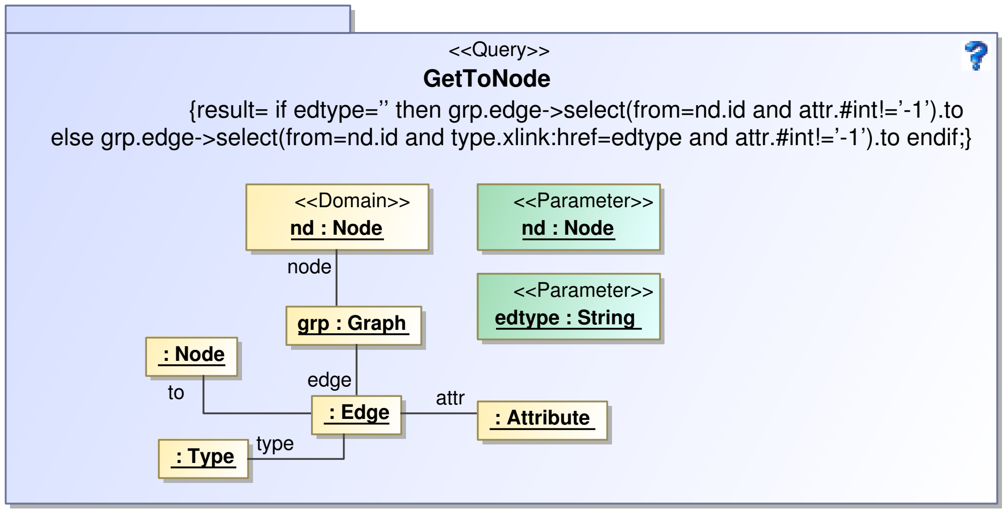

<<Query>>
GetToNode
{result= if edtype='' then grp.edge->select(from=nd.id and attr.#int!='-1').to
else grp.edge->select(from=nd.id and type.xlink:href=edtype and attr.#int!='-1').to endif;}
<<Domain>>
nd : Node
<<Parameter>>
nd : Node
node
grp : Graph
<<Parameter>>
edtype : String
: Node
edge
to
: Edge
attr
: Attribute
: Type
type

Figure 23: Get destination nodes of a node

B.3 Functions

```
less = if op='LESS' and v0 < v1 then 'true' else '' endif;
noless = if op='LESS' and v0 > v1 then 'false' else '' endif;
grt = if op='GREATER' and v0 > v1 then 'true' else '' endif;
nogrt = if op='GREATER' and v0 < v1 then 'false' else '' endif;
eq= if op='EQUAL' and v0 = v1 then 'true' else '' endif;
noeq= if op='EQUAL' and v0 != v1 then 'false' else '' endif;

result = less + noless + grt + nogrt + eq + noeq;
```

Figure 24: Function **CalcuLogic**(v0: Integer, v1: Integer, op:String) : String

```
add = if op='#Add' then v0 + v1 + 0 else 0 endif;
sub = if op='#Sub' then v0 − v1 else 0 endif;
mul = if op='#Mul' then v0 ∗ v1 else 0 endif;
div = if op='#Div' then v0 / v1 else 0 endif;

result = add + sub + mul + div + 0;
```

Figure 25: Function **CalcuMatch**(v0: Integer, v1: Integer, op:String):Integer

C Transformation for instruction selection

• **Transformation configuration:** name : *TTC_InstructionSelection*, source : *Intermediate*, sourceKey : *id*, sourceName : *srcgrp*, target: *Intermediate*, targetKey:*id*, targetName : *trggrp*.

C.1 QVTR relations

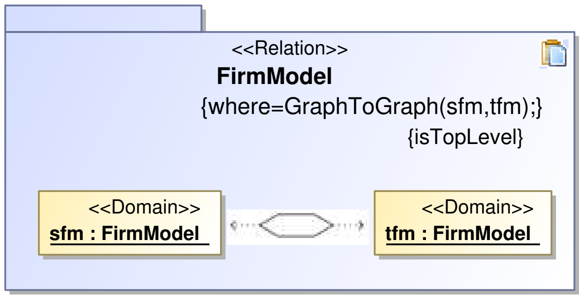


Figure 26: Starting top level relation

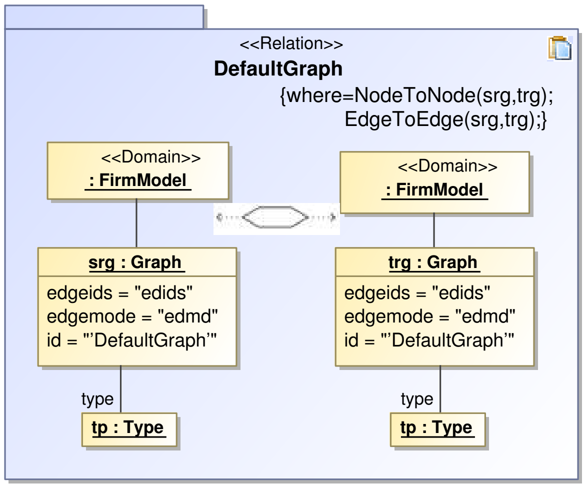


Figure 27: Cope with default graph (relName : *GraphToGraph*)

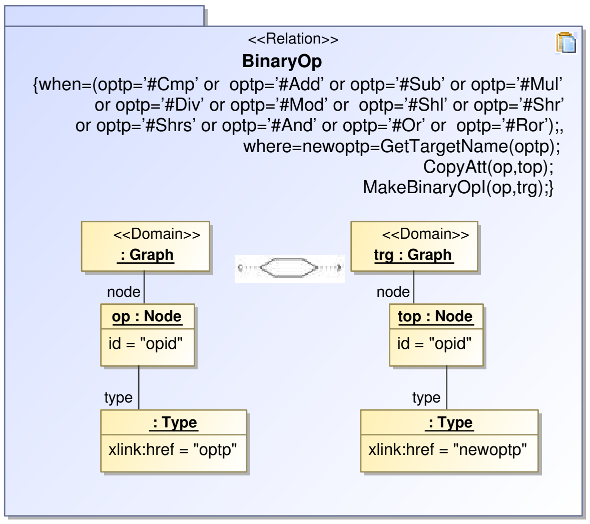

<<Relation>>
BinaryOp
{when=(optp='#Cmp' or optp='#Add' or optp='#Sub' or optp='#Mul'
or optp='#Div' or optp='#Mod' or optp='#Shl' or optp='#Shr'
or optp='#Shrs' or optp='#And' or optp='#Or' or optp='#Ror');,
where=newoptp=GetTargetName(optp);
CopyAtt(op,top);
MakeBinaryOpl(op,trg);}
<<Domain>>
: Graph
<<Domain>>
trg : Graph
node
op : Node
id = "opid"
node
top : Node
id = "opid"
type
: Type
xlink:href = "optp"
type
: Type
xlink:href = "newoptp"

Figure 28: Select and cope with binary operations (relName: *NodeToNode*)

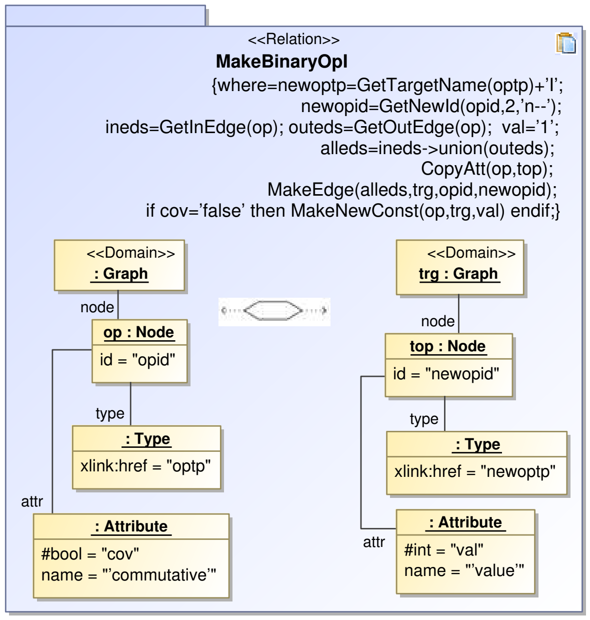


Figure 29: Create the TargetOpl node

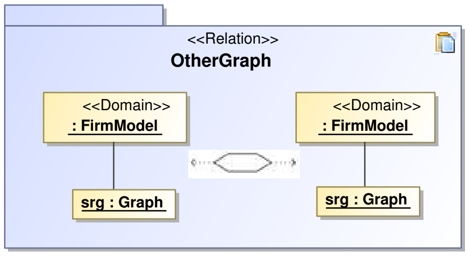


Figure 30: Copy graphs other than the default one (relName: *GraphToGraph*)

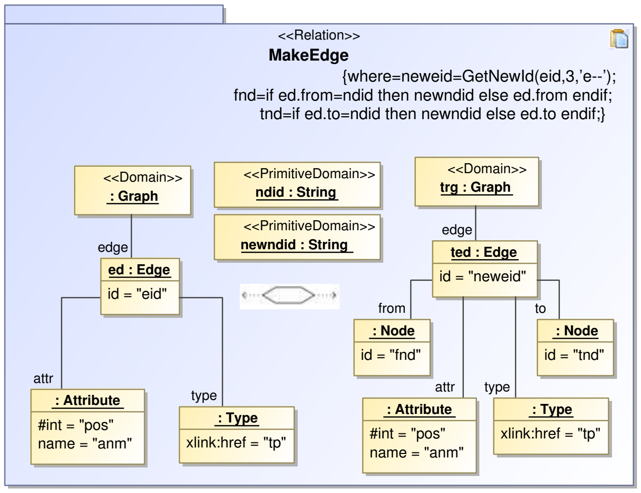


Figure 31: Create edges for the TargetOpI node

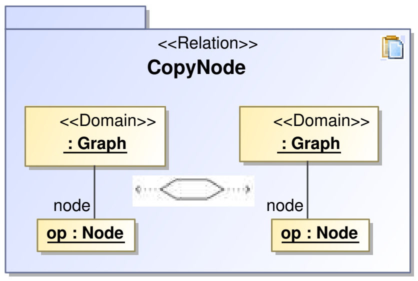


Figure 32: Copy node of other type (relName: *NodeToNode*)

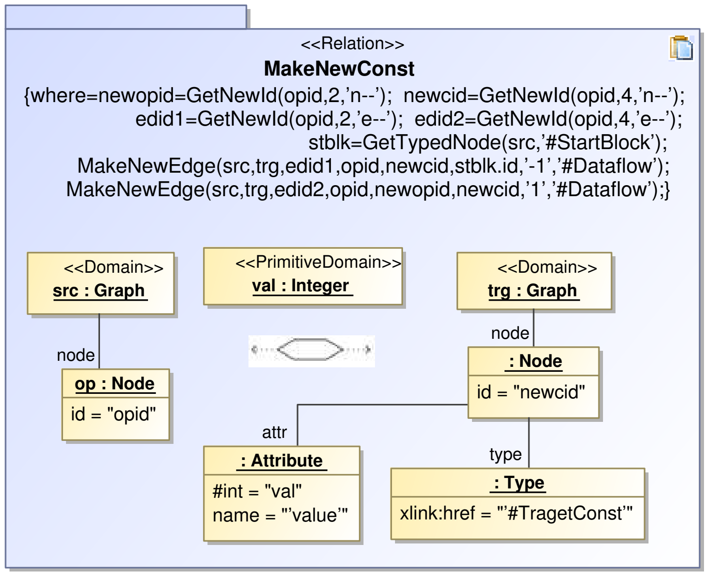
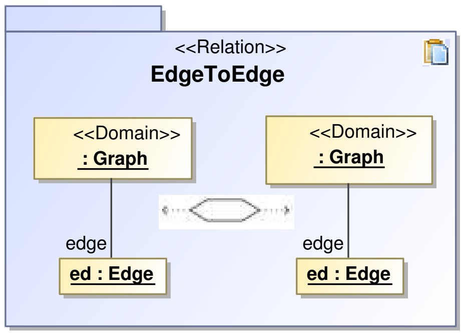
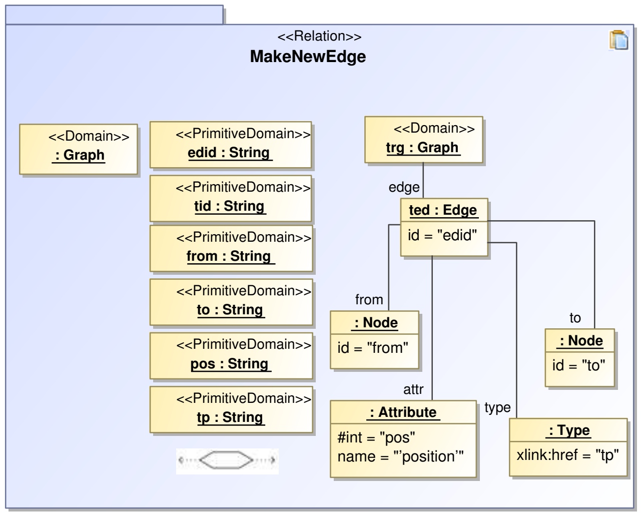

<<Relation>>
MakeNewConst
{where=newopid=GetNewId(opid,2,'n--'); newcid=GetNewId(opid,4,'n--');
edid1=GetNewId(opid,2,'e--'); edid2=GetNewId(opid,4,'e--');
stblk=GetTypedNode(src,'#StartBlock');
MakeNewEdge(src,trg,edid1,opid,newcid,stblk.id,'-1','#Dataflow');
MakeNewEdge(src,trg,edid2,opid,newopid,newcid,'1','#Dataflow');}
<<Domain>>
src : Graph
<<PrimitiveDomain>>
val : Integer
<<Domain>>
trg : Graph
node
op : Node
id = "opid"
node
: Node
id = "newcid"
attr
: Attribute
#int = "val"
name = "'value'"
type
: Type
xlink:href = "'#TragetConst'"

Figure 33: Create a new const node for the TargetOpI node





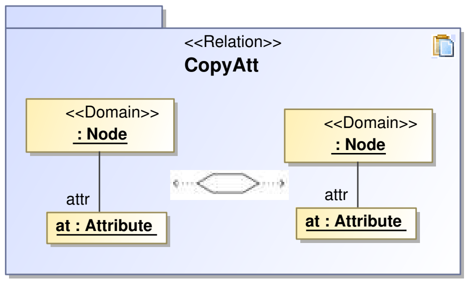


Figure 36: Copy attribute of node

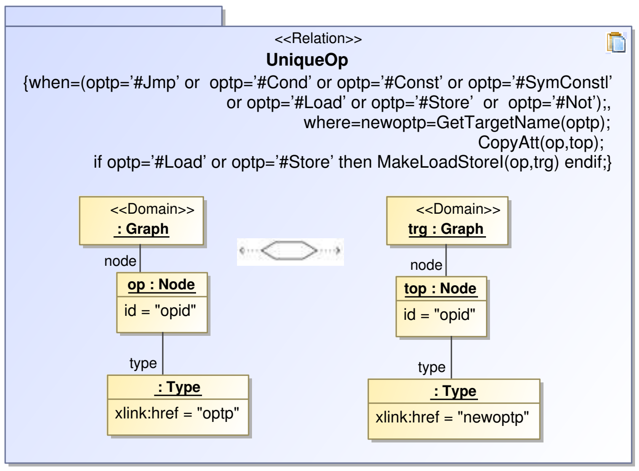

<<Relation>>
UniqueOp
{when=(optp='#Jmp' or optp='#Cond' or optp='#Const' or optp='#SymConstl' or optp='#Load' or optp='#Store' or optp='#Not');,
where=newoptp=GetTargetName(optp);
CopyAtt(op,top);
if optp='#Load' or optp='#Store' then MakeLoadStoreI(op,trg) endif;}
<<Domain>>
: Graph
node
op : Node
id = "opid"
type
: Type
xlink:href = "optp"
<<Domain>>
trg : Graph
node
top : Node
id = "opid"
type
: Type
xlink:href = "newoptp"

Figure 37: Select and cope with other operations (relName: *NodeToNode*)

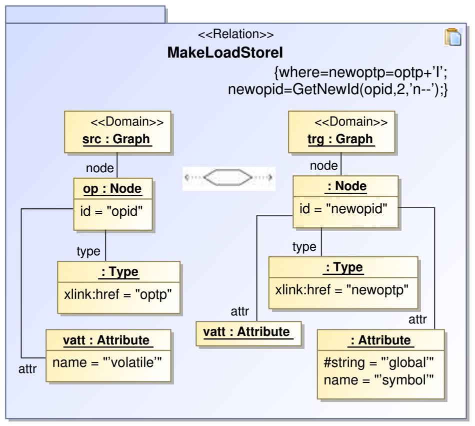


Figure 38: Create LoadI or StoreI node

C.2 Queries and Functions

All queries have same definitions as in transformation local optimizations:

- **GetInEdge** (Fig. 17), **GetOutEdge** (Fig. 18), **GetOwnerBlock** (Fig. 20)
- **GetToData** (Fig. 22), **GetTypedNode** (Fig. 21)

pos1=pos+1;
p1=substring(in,1,pos);
p2=substring(in,pos1);
result=sufix+p2+p1;

Figure 39: Function **GetNewId**(in: String, pos: Integer, sufix:String)

nm=substring−after(op,'#');
result='#'+'Target'+nm;

Figure 40: Function **GetTargetName**(op : String)