OCL as Expression Language in an Action Semantics Surface Language

Stefan Haustein and Jörg Pleumann

Computer Science Dept. VIII/X
University of Dortmund
Germany
{stefan.haustein,joerg.pleumann}@udo.edu

Abstract. With the specifiaction of Action Semantics in UML 1.5, the OMG has layed ground to manipulating object diagrams in a formal way, which is a necessary prerequisite for QVT. In QVT, of course the manipulations take place at M1 level instead of M0, but due to the architecture of UML, the same mechanisms can simply be reused. Unfortunately, the Action Semantics specification does not mandate a surface language, limiting its practical application. Due to the high overlap with the Object Constraint Language, in this article we propose a surface language that is based on and aligned with OCL.

1 Introduction

In earlier versions of the UML standard, textual descriptions were used to capture the behavior of operations in an informal way. However, this form of description is not accessible to tools like theorem provers, model-based simulation, and code generators [5]. To overcome this problem, UML 1.5 introduced the specification of operations using Action Semantics (AS) [7, chapter 2]. Action Semantics is a framework for the formal description of programming languages [6].

The Action Semantics proposal accepted by the OMG formalizes different categories of actions [1] for inclusion in UML 1.5 and 2.0:

Control Structures are used to model loop and branch structures.
Read and Write Actions are used to access the values of object properties and to create new instances or to delete existing instances.
Computation Actions transform a set of input values to produce a set of output values without side effects on other parts of the system.
Collection Actions permit the parallelizable application of an action to a set of data elements.
Messaging Actions trigger asynchronous or synchronous actions such as state machine transitions or method invocations with a return value.
Compositional Actions model iterations and conditionals.
Jump Actions allow deviations from the main path of control flow in iterations similar to the *break statement known from many programming languages.*

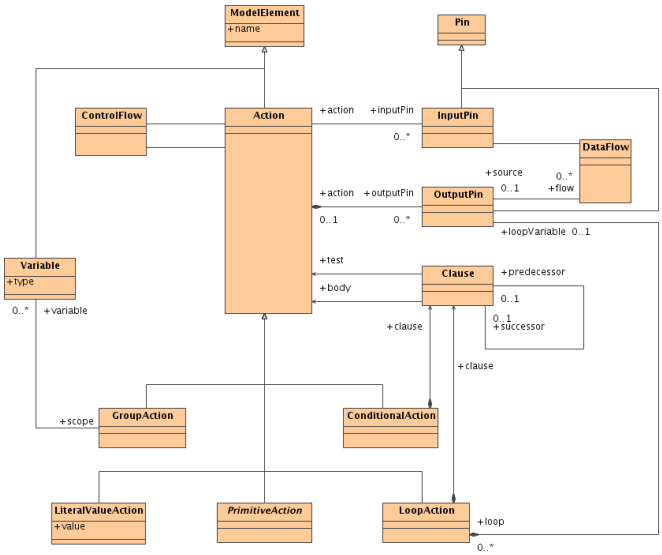

Fig. 1. Control action overview

Following the general spirit of UML, those actions are modeled as UML classes. Figure 1 shows an overview of the classes describing control structures in the Action Semantic framework. The other categories are described in a similar way. Each action has a set of input and output *pins*. Output pins can be connected to one or more input pins of other actions, creating sequential dependencies. Alternatively to the pin mechanism, the class *ControlFlow* can be used to explicitly enforce an order of execution. The class diagrams of the Action Semantics are roughly similar to the grammar specifying the abstract syntax of a programming language. Naturally, object diagrams could be used to capture the equivalent of the abstract syntax tree of a program, altough even for simple expressions the diagrams become extremely verbose. What is missing in this system is a concrete syntax, which is called surface language in the Action Semantics specification.

2 Action Semantics and the Object Constraint Language

The UML Action Semantics recognizes the need for an Action Semantics surface language, but does not recomend a specific one. The specification contains a set of informal mappings to action languages used in different UML tools, namely the Action Specification Language (ASL) [3], the BridgePoint Action Language (AL) [9], the Kabira Action Semantics (Kabira AS) [2], and the action language subset of the Specification and Description Language (SDL), an international standard widely used in the telecommunication industry [10].

All those languages rely on a syntax that is incoherent with the existing UML expression language, the Object Constraint Language (OCL). Actually, large parts of the Action Semantics specification duplicates functionality that is already covered by the OCL, such as

- Navigation and read access to properties
- Computation
- Calls to query operations
- Collection operations

The great overlap of the model access constructs defined in the AS and OCL specifications suggests that using two completely different syntaxes may be inappropriate and confusing; one would expect a surface language that leverages existing OCL knowledge and infrastructure. Naturally, due to the side effect free nature of OCL, OCL cannot cover actions such as write actions or calls to non-query operations, but using OCL for the parts covered would mean a significant improvement over the current situation.

It seemes quite straight-forward to build an action surface language that is based on OCL expressions, without tainting OCL itself with side effects. Anneke Kleppe and Jos Warner suggest an action clause as an extension to OCL that would fit nicely with the declarative nature of OCL [4], but for the desired purpose a fully operational solution is required. Since the OCL is a subset of the AS, there are two options for building an action surface language based on OCL:

1. Map all OCL constructs to actions, then add new syntax constructs for actions that are required, but not covered.
2. Embed OCL expressions in new syntax constructs for actions.

The first option requires a complete mapping of the abstract OCL syntax to actions. This would mean to give up declarative semantics in OCL, or to have two flavours of OCL with different specifications that would need to be aligned carefully.

The second option can be implemented by referring to the existing OCL surface language, without modifying it, maintaining a clean syntactical separation between plain queries and actions that may influence the system state. This approach keeps the interface between bothe languages minimal and allows to keep both languages relatively separate, not tainting OCL by introducing side effects to OCL itself. Since this approach seemed the more promising one, we have implemented it in our Infolayer system.

3 ASOQ

The Infolayer system is basically an UML runtime environment that interprets class diagrams and state charts as a Web application. It can be seen as an implementation of a variant of MDA that does not compile PIMs, but interprets

them instead. In this approach, the transformation from the PIM to the PSM is handled implicitly by a model-driven runtime (MDR) environment. Where MDA potentially transforms object-oriented concepts to non object-oriented ones (as in the case of the relational database), our MDR implements selected parts of the UML metamodel and interprets them for a given application domain.

In the system, OCL is used to implement user defined operations without side effects. To implement operations that are not free of side effects, we have implemented a language termed *Action Semantics Surface Language based on OCL Queries* (ASOQ), following the ideas outlined in the previous section.

Syntax constructs needed to be created only for the functionality that is not already present in OCL, namely composite actions, write actions, and messaging actions. We tried to align the new constructs with existing OCL syntax:

- The OCL *if–then–else* structure includes *endif* as a specific end marker, where other languages such as C and PASCAL use an explicit block structure, marked by curly brackets or special keywords such as *begin* and *end*. For consistency, implicit blocks and end markers specific to the control structure are used in ASOQ also for loops (while–do–enddo).
- '=' is used mainly as comparison operator in OCL; when used in assignments it may be paired with a colon and a type declaration. Thus, using ':=' for property and variable assignments seems consistent with OCL.
- OCL uses dots ('.') to separate parts of path expressions. In ASOQ, we will use the exclamation mark ('!') to indicate an operation with side effects at the end of a path expression.
- The OCL *let...in* declaration block and the *if–then–else* structure can be reused in the ASOQ with an identical syntax to build group actions and conditional actions.

Before we can define the details of the syntax, we need a construct to integrate arbitrary OCL expressions in the Action Semantics framework. For this purpose, we create a class *OclAction* that inherits from *PrimitiveAction*, but has a link to an *OclExpression* object, defined in the OCL specification. The new class allows us to embed OCL expressions in chains of actions. It can have at most one *result* output pin, which delivers the result of the evaluation of the expression. The upper part of figure 2 shows the *OclAction* class. The *isReadOnly* property of the *OclAction* is always true, since OCL expressions cannot have any side effects. Variable references can be handed from actions to the OCL parser by pre-initializing the OCL environment accordingly. Full read access to the System state is already available in OCL.

The OCL specification includes a mapping from the concrete syntax to the abstract syntax in the form of a full attribute grammar. The full attribute grammar contains not only the syntax definitions, but also a specification of the resulting abstract syntax tree. For consistency with the OCL specification, we use the same formalism to specify the ASOQ syntax. This allows us also to include some OCL constructs by reference.

The appendix contain the syntax specification of the ASOQ building blocks that complement the OCL to a Action Semantics surface language:

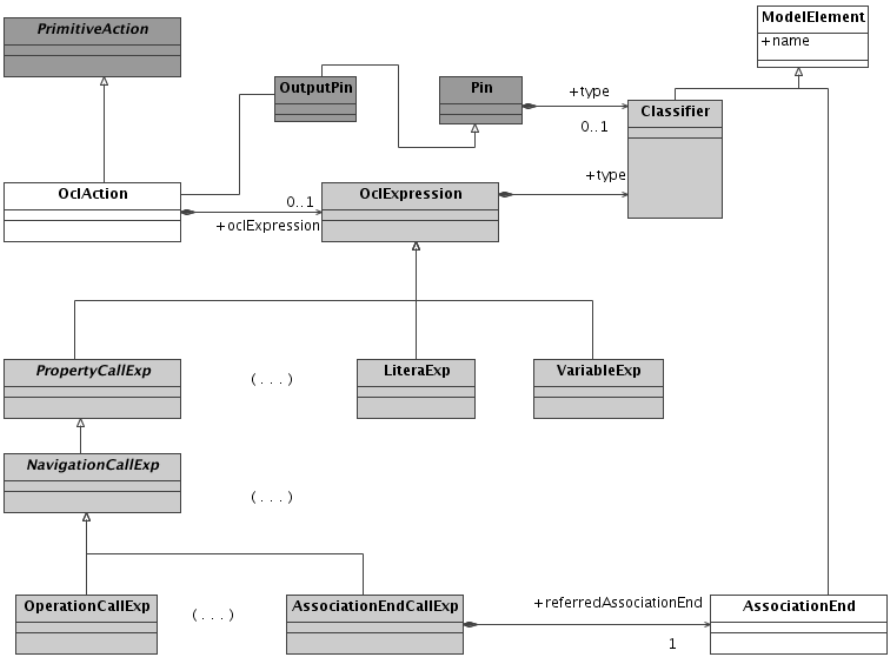

Fig. 2. OclAction and related classes. Classes contained in the Action Semantics specification are dark gray, classes from the OCL specification are light gray. The new class *OclAction* and UML core classifiers are white.

– A simple block struckture including variables and statements
– The OclAction allowing to embed OCL construct
– An if-then-else conditional structure
– A while loop structure
– Non-query method invocations
– Variable assignments
– Property assignments

4 Conclusion and Outlook

The introduction of the Action Semantics framework into the UML has provided a means of describing UML actions and operation implementations in a formal way. This is an important building block for model transformations and code generation in the context of MDA and QVT. The Action Semantics specification only describes abstract syntax and thus stays largely language-independent. Yet, for practical purposes, a concrete syntax is required, too. We have designed such a language – named ASOQ – based on OCL, and we have been able to minimize the specification effort by integrating the existing OCL specification into this language. An implementation of ASOQ is used in the Infolayer system [8], where it serves for manipulating the system state at the M0 level. However, our positive experience with the language suggests that it might also be used at M1 level in the context of a transformation language like QVT.

References

1. Action Semantics for the UML, August 2001. OMG ad/2001-08-04.
2. Kabira Technologies, Inc. Kabira Action Semantics. http://www.kabira.com.
3. Kennedy Carter Ltd. Action Specification Language (ASL). http://www.kc.com.
4. Anneke Kleppe and Jos Warmer. Extending OCL to include actions. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 278–293. Springer, 2000.
5. Stephen J. Mellor, Steve Tockey, Rodolphe Arthaud, and Philippe LeBlanc. Software-platform-independent, precise action specifications for UML. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 281–286, 1998.
6. Peter D. Mosses. Theory and practice of action semantics. Technical Report BRICS RS-96-53, Department of Computer Science, University of Aarhus, Ny Munkegade, building 540, DK-8000 Aarhus C, Denmark, 1996.
7. Object Management Group (OMG). Unified Modeling Language (UML) 1.5 Specification. http://www.omg.org/cgi-bin/doc?formal/03-03-01, 2003.
8. Jörg Pleumann and Stefan Haustein. A model-driven runtime environment for web application. In *UML Conference 2003*, LNCS. Springer Verlag, 2003.
9. Project Technology, Inc. BridgePoint Action Language (AL). http://www.projtech.com.
10. International Telecommunication Union. Specification and description language (SDL). Technical Report Z.100, ITU-T, 1999.

A Appendix: ASOQ Grammar

A.1 Blocks, Variables, and Statements

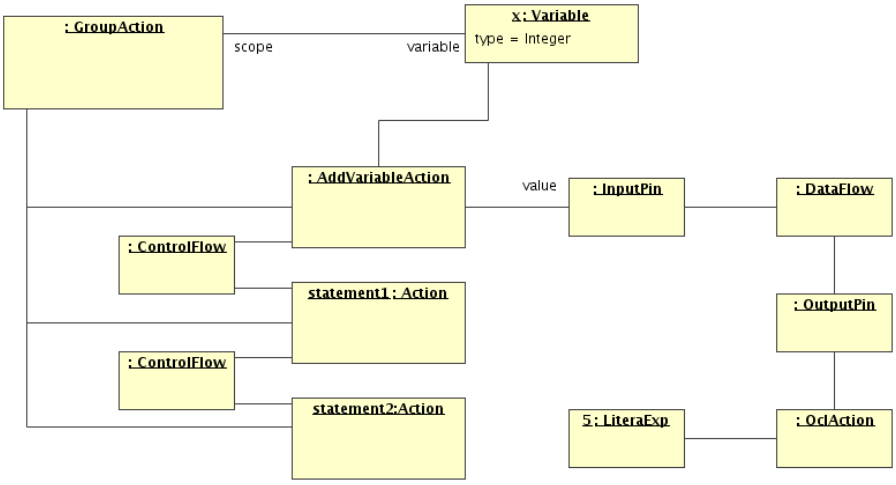

Fig. 3. Mapping of a *block* to a *GroupAction* object

The top level concept of ASOQ is a block, consisting of a list of variable declarations, followed by a sequence of statements. Figure 3 shows an action object diagram for the following block:

```
let
  x = 5
in
-- Statement 1;
-- Statement 2
```

In the following annotated syntax definition, *SimpleNameCS* and *OclExpressionCS* from the OCL Grammar are referenced. *OclActionCS* simply wraps the *OclExpression* abstract syntax tree in an *OclAction* object. The variable declaration syntax cannot be simply taken from the OCL specification in order to allow an initialization with a newly created object. The following rules spend most space for the correct chaining of actions.

BlockCS ::= StatementListCS

```
StatementListCS.env = BlockCS.env
BlockCS.ast : GroupAction
BlockCS.ast.subaction = StatementListCS.ast
```

BlockCS ::= LetBlockCS

```
LetBlockCS.env = BlockCS.env
BlockCS.ast : GroupAction
BlockCS.ast.subaction = LetBlockCS.ast
```

LetBlockCS ::= 'let' AsoqVariableDeclarationCS LetBlockSubCS

```
LetBlockSubCS.env =
  LetBlockCS.env.nestedEnvironment().addElement(
    AsoqVariableDeclarationCS.variable.name,
    AsoqVariableDeclarationCS.variable,
    false)
LetBlockCS.ast = LetBlockSubCS.ast->prepend(
  AsoqVariableDeclarationCS.ast)
LetBlockCS.ast->first().consequent : ControlFlow
LetBlockCS.ast->first().consequent.sucessor =
  LetBlockSubCS.ast->first()
```

LetBlockSubCS ::= ',' AsoqVariableDeclarationCS LetBlockSubCS[2]

```
LetBlockSubCS[2].env =
  LetBlockSubCS.env.nestedEnvironment().addElement(
    AsoqVariableDeclarationCS.variable.name,
    AsoqVariableDeclarationCS.variable,
    false)
LetBlockSubCS.ast = LetBlockSubCS[2].ast->prepend(
  AsoqVariableDeclarationCS.ast)
LetBlockSubCS.ast->first().consequent : ControlFlow
LetBlockSubCS.ast->first().consequent.sucessor =
  LetBlockSubCS[2]->first()
```

LetBlockSubCS ::= 'in' StatementListCS

```
StatementList.env =
  LetBlockSubCS.env.nestedEnvironment().addElement(
    AsoqVariableDeclarationCS.variable.name,
    AsoqVariableDeclarationCS.variable,
    false)
LetBlockSubCS.ast = StatementListCS.ast->prepend(
  AsoqVariableDeclarationCS.ast)
LetBlockSubCS.ast->first().consequent : ControlFlow
LetBlockSubCS.ast->first().consequent.sucessor =
  StatementList->first()
```

AsoqVariableDeclarationCS ::= simpleNameCS : typeCS = AsoqExpressionCS

```
AsoqExpression.env = AsoqVariableDeclarationCS.
VariableCS.ast : WriteVariableAction
VariableCS.ast.variable : Variable
VariableCS.ast.variable.name : String = simpleNameCS
```

```
VariableCS.ast.variable.in : InputPin
VariableCS.ast.variable.in.flow : DataFlow
VariableCS.ast.variable.in.flow.source : OutputPin
VariableCS.ast.variable.in.flow.source.action =
  AsoqExpressionCS
```

StatementListCS ::= StatementCS

```
Statement.env = StatementListCS.env
StatementListCS.ast = Sequence{StatementCS.ast}
```

StatementListCS ::= StatementCS ';' StatementListCS[2]

```
StatementListCS.ast =
  StatementListCS[2].ast->prepend(StatementCS.ast)
StatementCS.ast.consequent : ControlFlow
StatementCS.ast.consequent.successor =
  StatementListCS[2].ast->first
```

StatementCS ::= BlockCS

```
BlockCS.env = StatementCS.env
StatementCS.ast = Block.ast
```

StatementCS ::= IfStatementCS

```
IfStatementCS.env = StatementCS.env
StatementCS.ast = IfStatementCS.ast
```

StatementCS ::= WhileStatementCS

```
WhileStatementCS.env = StatementCS.env
StatementCS.ast = WhileStatementCS.ast
```

StatementCS ::= AssignmentCS

```
AssignmentCS.env = StatementCS.env
StatementCS.ast = AssignmentCS.ast
```

StatementCS ::= AsoqExpressionCS

In order to avoid ambiguities with OCL *let* and *if* constructs, all other statement rules must take precedence to this one.

```
AsoqStatementCS.env = StatementCS.env
StatementCS.ast = AsoqExpressionCS.ast
```

A.2 OclAction

A simple rule simplifies the integration of OCL expressions in the action syntax:

OclActionCS ::= OclExpressionCS

```
OclExpressionCS.env = OclActionCS.env
OclActionCS.ast : OclAction
OclActionCS.ast.expression = OclExpressionCS.ast
```

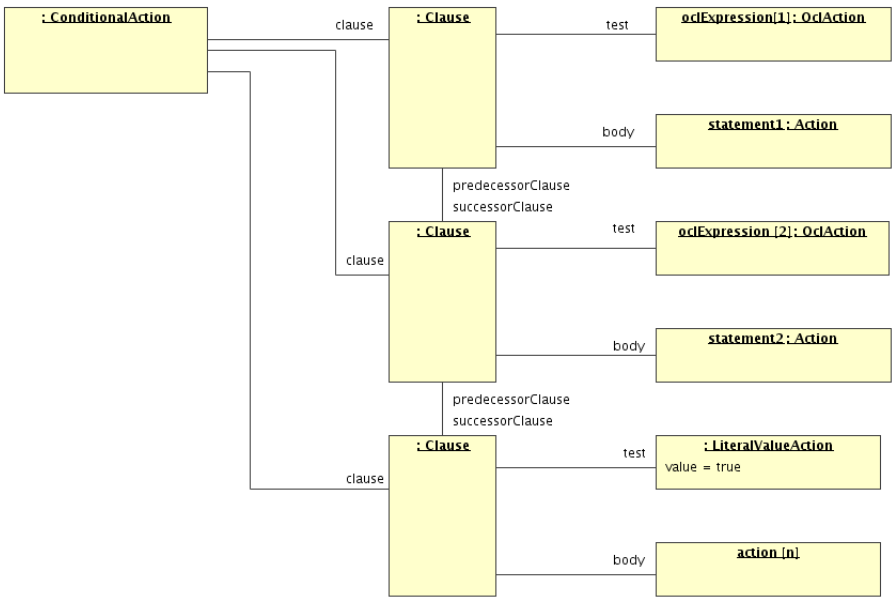

Fig. 4. Mapping of an if statement to a *GroupAction* object

A.3 if-then-else

The OCL already contains an *if-then-else* control structure, but at ASOQ statement level it is not necessary that a value is returned, thus the *else* part may be omitted. To avoid deep nesting for multi-way decisions, while still keeping the syntax consistent with OCL, we add an *elseif* construct. Other languages such as C or PASCAL do not need this because they use an explicit general block structure—instead of implicitly opening a block that must be terminated with *endif*.

An example code snippet, matching the multi-way decision sample of the action semantics specification, looks as follows:

```
if factor = 1 then
  -- Some action 1
else if facror = 2 then
  -- Some action 2
else
  -- Some action 3
endif
```

The resulting action semantic object diagram is depicted in figure 4.

IfStatementCS ::= 'if' ClauseCS 'endif'

```
ClauseCS.env = IfStatementCS.env
IfStatementCS.ast : ConditionalAction
IfStatementCS.ast.clause = Set{ClauseCS.ast}
```

IfStatementCS ::= 'if' ClauseCS ElseCS 'endif'

```
IfStatementCS.ast : ConditionalAction
IfStatementCS.ast.clause =
  ElseifCS.ast->prepend(ClauseCS)
ClasuseCS.ast.successorClause = ElseifCS->first()
```

ClauseCS ::= OclActionCS 'then' BlockCS

```
ClauseCS.ast : Clause
ClauseCS.ast.test = OclActionCS.ast
ClauseCS.ast.body = BlockCS.ast
```

ElseCS ::= 'else' DefaultClauseCS

```
ElseCS.ast = Sequence{DefautlClauseCS.ast}
```

ElseCS ::= ElseifCS 'else' DefaultClauseCS

```
ElseCS.ast = ElseifCS.ast->append(DefautlClauseCS.ast)
DefaultClauseCS.ast.predecessorClause =
  ElseifCS.any(successorClause->isEmpty)
```

DefaultClauseCS ::= BlockCS

```
DefaultClauseCS.ast : Clause
DefaultClauseCS.ast.test : LiteralActionCS
DefaultClauseCS.ast.test.value = true
DefaultClauseCS.ast.body = BlockCS.ast
```

ElseifCS ::= 'elseif' ClauseCS

```
ClauseCS.env = ElseifCS
ElseifClauseCS.ast = Sequence{ClauseCS}
```

ElseifCS ::= 'elseif' ClauseCS Elseif[2]CS

```
ClauseCS.env = ElseifCS.env
ElseIfCS[2].env = ElseifCS.env
ElseifCS.ast = Elseif[2]CS.ast->prepend(ClauseCS.ast)
ClauseCS.ast.successorClause = ElseifCS[2].ast->first()
```

A.4 while

A while loop is fortunately quite straightforward to construct:

WhileStatementCS ::= 'while' OclActionCS 'do' BlockCS 'enddo'

```
WhileStatementCS.ast : LoopAction
WhileStatementCS.ast.clause : Clause
WhileStatementCS.ast.clause.test = OclExpressionCS.ast
WhileStatementCS.ast.clause.body = BlockCS.ast
```

A.5 Method Invocations and ASOQ Expressions

Non-query Operations with and without parameters can be invoked just like query operations are invoked in OCL. All parameters must be OCL expressions, it is not possible to nest invocation actions. An optional path to the operation may be provided as an OCL expression. If so, the operation invocation must be separated from the path expression with an exclamation mark. An actual ASOQ code snippet may look as follows:

```
Publication.allIstances()->one
  (title='Cold Fusion')!vote()
```

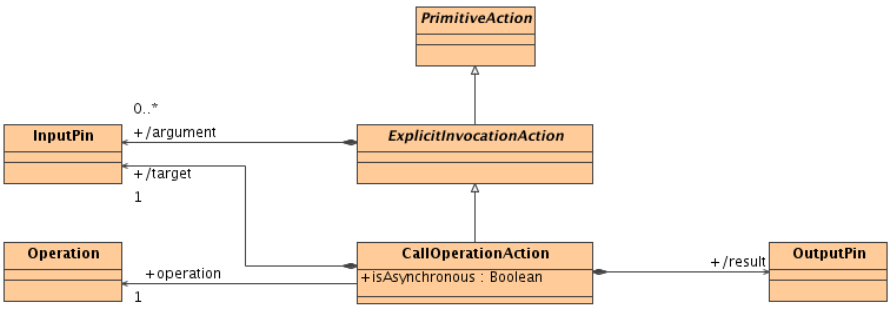

Fig. 5. Messaging actions referenced in ASOQ

Figure 5 shows the relevant parts of the action semantics used in the following syntax definiton.

AsoqExpressionCS ::= simpleNameCS ArgumentsCS

```
ArgumentCS.env = AsoqExpressionCS.env
AsoqExpressionCS.ast : CallOperationAction
AsoqExpressionCS.ast.argument : Sequence(InputPin)
AsoqExpressionCS.ast.argument->size() =
  ArgumentCS.ast->size()
AsoqExpressionCS.ast.argument->forAll
  (i|i.flow : Dataflow)
Sequence{1..ArgumentCS->size()}->forAll(i|
  AsoqExpressionCS.ast.argument->at(i).flow.source =
    ArgumentCS->at(i).result)
AsoqExpressionCS.ast.operation =
  env.lookUpImplicitNqOperation(
    simpleNameCS.ast,
    ArgumentCS.ast->collect(result.type))
```

AsoqExpressionCS ::= OclActionCS '!' simpleNameCS ArgumentsCS

```
ArgumentCS.env = AsoqExpressionCS.env
AsoqExpressionCS.ast : CallOperationAction
AsoqExpressionCS.ast.target = OclAction
AsoqExpressionCS.ast.target : InputPin
AsoqEspressionCS.ast.target.flow : DataFlow
AsoqExpressionCS.ast.target.flow.source =
  OclActionCS.result
AsoqExpressionCS.ast.operation =
  OclActionCS.result.type.lookUpNqOperation(
    simpleNameCS.ast,
    ArgumentCS.ast->collect(result.type))
AsoqExpressionCS.ast.argument : Sequence(InputPin)
AsoqExpressionCS.ast.argument->size() =
  ArgumentCS.ast->size()
AsoqExpressionCS.ast.argument
  ->forAll(i|i.flow : Dataflow)
Sequence{1..ArgumentCS->size()}->forAll(i|
  AsoqExpressionCS.ast.argument->at(i).flow.source =
    ArgumentCS->at(i).result)
```

AsoqExpressionCS ::= OclActionCS

```
OclActionCS.env = AsoqExpressionCS.env
AsoqExpressionCS.ast = OclActionCS.ast
```

ArgumentsCS ::= '(' ')'

```
ArgumentCS.ast : Sequence(OclAction)
ArgumentCS.ast = Sequence{}
```

ArgumentListCS ::= OclActionCS

```
OclActionCS.env = ArgumentListCS.env
ArgumentListCS.ast = Sequence{OclActionCS.ast}
```

ArgumentListCS ::= OclActionCS ',' ArgumentListCS[2]

```
OclActionCS.env = ArgumentListCS.env
ArgumentListCS[2].env = ArgumentListCS.env
ArgumentListCS.ast =
  ArgumentListCS[2].ast->prepend(OclActionCS.ast)
```

A.6 Assignments

The action semantics classes modeling assignments are depicted in figure 6. Variable assignments are simple to handle. It is only required to look up the name in the environment and then construct an according *WriteVariableAction*:

VariableAssignmentCS ::= SimpleNameCS ':=' AsoqExpressionCS

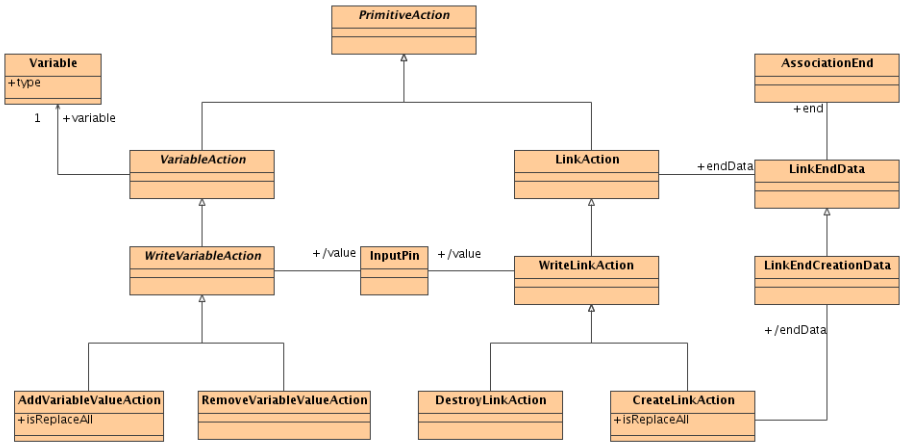

Fig. 6. Messaging actions referenced in ASOQ

```
SimpleNameCS.env = VariableAssignmentCS.env
AsoqExpressionCS.env = VariableAssignmentCS.env
VariableAssignmentCS.ast : AddVariableValueAction
VariableAssignmentCS.ast.isReplaceAll = true
VariableAssignmentCS.ast.variable =
  env.lookUpASVariable(SimpleNameCS)
VariableAssignmentCS.ast.value = InputPin
VariableAssignmentCS.ast.value.flow = DataFlow
VariableAssignmentCS.ast.value.flow.source =
  AsoqExpressionCS.outputPin
```

A.7 Property Assignments

Property assignments, where the target may be an arbitrary complex *OclExpression* are more difficult to construct. It is necessary to extract the property from the OCL *AssociationEndCallCS* in order to link an *OclAction* from the source expression of the property. A *CreateLinkAction* uses the output pin of the *OclAction* as input pin.

PropertyAssignmentCS ::= AssociationEndCallCS ':=' AsoqExpresisonCS

```
AssociationEndCallCS.env = PropertyAssignmentCS.env
AsoqExpressionCS.env = PropertyAssignmentCS.env
PropertyAssignmentCS.ast : CreateLinkAction
PropertyAssignmentCS.ast.endData : LinkEndData
PropertyAssignmentCS.ast.endData.end =
  AssociationEndCallCS.referredAssociationEnd
PropertyAssignmentCS.ast.endData.value : InputPin
```

```
PropertyAssignmentCS.ast.endData.value.flow = DataFlow
PropertyAssignmentCS.ast.endData.value.flow.source =
  OutputPin
PropertyAssignmentCS.ast.endData.value.flow.source
  .action = OclAction
PropertyAssignmentCS.ast.endData.value.flow.source
  .action.expression = AssociationEndCallCS.source
PropertyAssignmentCS.ast.isReplaceAll = true
PropertyAssignmentCS.ast.value = InputPin
PropertyAssignmentCS.ast.value.flow = DataFlow
PropertyAssignmentCS.ast.value.flow.source =
  AsoqExpressionCS.outputPin
```