Enabling OCL and fUML Integration by Transformation

Massimo Tisi[1], Frédéric Jouault[2(✉)], Zied Saidi[1], and Jérome Delatour[2]

[1] AtlanMod team (Inria, Mines Nantes, LINA), Nantes, France
{massimo.tisi,zied.saidi}@mines-nantes.fr
[2] TRAME team (ESEO), Angers, France
{frederic.jouault,jerome.delatour}@eseo.fr

Abstract. Until the recent adoption of fUML, UML has lacked standard execution semantics. However, parts of UML models have always been executable: OCL expressions. They may notably be used to express contracts (i.e., invariants, pre- and post-conditions), to define side-effect free operations, and to specify how to compute derived features. Nonetheless, although fUML is partly inspired by OCL (notably for primitive behaviors), its specification does not consider interoperability with OCL expressions. Moreover, the semantics of OCL is specified independently of (f)UML, and their implementations are separate execution engines, hampering all global activities (e.g., analysis, optimization, debugging). This paper explores a possible integration approach of OCL and fUML: by transforming (i.e., compiling) OCL expressions into fUML activities we obtain a unified executable model. With this approach, operations specified in OCL can be called, and getters can be generated for derived features. Preconditions (resp. postconditions) can be automatically executed before (resp. after) the execution of their contextual operations. A precise semantics for invariant evaluation can be specified in fUML. Thanks to this work, OCL may also be seen as a functional counterpart to Alf.

1 Introduction

Foundational UML (fUML) [1] is the ongoing effort by the Object Management Group (OMG) for providing standard execution semantics to UML models. Executable models may be beneficial in Model-Driven Engineering (MDE) when a higher degree of precision is required at the modeling level (e.g., for critical systems), and for allowing users to simulate and analyze the system behavior before the actual implementation. fUML acts as the cornerstone of this vision, by providing, in its current version, standard execution semantics for a core subset of UML Activity Diagram and Class Diagram. Extensions are in progress for Statechart Diagram (Precise Semantics of UML State Machines, PSSM [2]) and Composite Structure Diagram (Precise Semantics of UML Composite Structures PSCS [3]). The Action Language For Foundational UML (Alf) [4] is a textual language that executes by translation towards fUML activities, and that is designed as a convenient alternative to complex diagrams.

© Springer International Publishing Switzerland 2016
A. Wąsowski and H. Lönn (Eds.): ECMFA 2016, LNCS 9764, pp. 156–172, 2016.
DOI: 10.1007/978-3-319-42061-5_10

Before the introduction of fUML, UML users could provide precise executable semantics only to a limited part of their models. Common practice was (and still is) to express executable contracts, side-effect free operations and computation of derived features by using the Object Constraint Language (OCL) [5], a purely functional language standardized by OMG. Moving to executable modeling by fUML/Alf does not lower the need for contracts, functional operations and derived features. However fUML does not include a built-in support for these tasks, that could constitute a viable alternative to OCL.

The specifications of OCL and fUML do not directly reference each other, and the interaction between the two languages passes through standard UML mechanisms: e.g., OCL expressions can call UML operations implemented in fUML activities and fUML activities can use UML value specifications computed by OCL expressions. Also the execution environment of the two languages is generally separated: OCL evaluators and fUML interpreters are separated tools, interacting at runtime on the same modeling platform (e.g., Java and the Eclipse Modeling Framework). This makes global optimization extremely difficult and hampers the possibilities of global analysis and simulation promised by the executable modeling approach.

In this paper we propose a mechanism for compile-time integration of OCL and fUML. We use a translational approach by providing a compiler able to transform a model including both OCL expressions and fUML activities into a pure fUML model. The resulting model is semantically equivalent to the original one, meaning that the OCL semantics for contracts, derived features, operation bodies, etc. has been implemented as fUML activities.

The paper contribution is twofold: (1) we provide a translation from OCL expressions to equivalent fUML activities; (2) we use fUML to provide a default semantics for under-specified parts of the OCL standard, e.g. timing of feature derivation. We also validate the feasibility of the approach by developing a proof-of-concept implementation of our mapping in the form of a compiler transformation, publicly available[1].

The paper is structured as follows: Sect. 2 introduces a running example in order to exemplify the problem of integrating OCL and fUML; Sect. 3 provides an intuitive look on our solution by describing its application to the example; Sect. 4 describes the mapping rules between OCL and fUML; Sect. 5 gives implementation details on our proof-of-concept compiler; Sect. 6 analyzes design decisions, limitations and alternative applications of the approach; Sect. 7 discusses related work and Sect. 8 concludes the paper.

2 Interaction Between fUML and OCL by Example

As an example of the joint use of OCL constraints and fUML behavior we adapt the Company example from the OCL specification [5, pp. 7–29]. The class diagram is presented in Fig. 1.

[1] https://github.com/atlanmod/OCL2fUML.

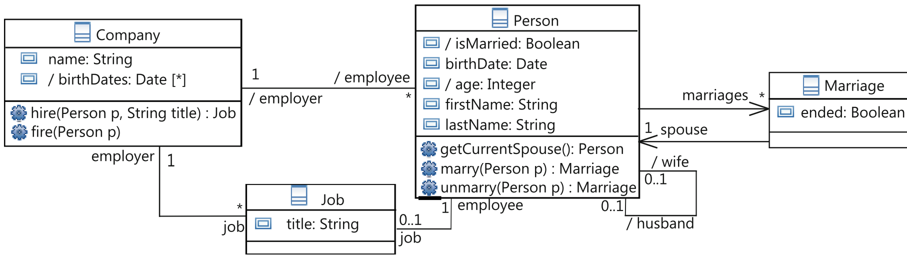

Fig. 1. Excerpt of the Company class diagram (adapted from [5, p. 7])

In this model a `Person` is described by her `firstName`, `lastName` and `birthDate`. The `age` of each `Person` is derived from her `birthDate`. A `Person` may be involved in one or more `Marriages`, but among them at most one is not currently `ended`[2]. To simplify access to marriage information, the `Person` class includes the `isMarried`, `husband` and `wife` derived features, and the `getCurrentSpouse()` operation whose precondition (i.e., the person is currently married) and function body (i.e., get the spouse of the only marriage that is not ended) are shown in Listing 1.1. Since the operation is free from side-effects its body can be expressed in OCL. According to the UML specification a call to `getCurrentSpouse()` returns the evaluation of the body only if the precondition is satisfied, otherwise the behavior is undefined.

Listing 1.1. getCurrentSpouse operation [5, p. 9]

```
context Person::getCurrentSpouse() : Person
  pre: self.isMarried = true
  body: self.marriages->select( m | m.ended = false ).spouse
```

The operations `marry(p: Person)` and `unmarry(p: Person)` are used to simplify model updates in case of marriage and divorce (avoiding the need to manually perform model element edits). Husbands and wives are adults, as specified by the OCL invariant in Listing 1.2. According to the OCL semantics, any update to the model, e.g., by the `marry` and `unmarry` operations, may trigger the re-verification of this invariant and the re-computation of the above-mentioned derived features. For instance, the evaluation of the invariant to false at the end of the operation execution results in an undefined behavior. The moment in which derived features are re-evaluated is not precisely specified by OCL, but computation will be performed before the subsequent access to the feature.

Listing 1.2. Wives and husbands are adults [5, p. 20]

```
context Person inv:
  (self.wife->notEmpty() implies self.wife.age >= 18) and
  (self.husband->notEmpty() implies self.husband.age >= 18)
```

Each `Person` may have a `Job` in a `Company`, which is identified by a `name` and cannot have less than 50 employees, as imposed by the invariant in Listing 1.3.

[2] For space reasons we can not provide here the code of all OCL contracts, derived features and fUML activities mentioned in this section.

Listing 1.3. Companies have at least 50 employees (adapted from [5, p. 20])

```
context Company inv:
  self.employee->size() >= 50
```

Derived features employer and employee simplify navigation between Companies and Persons. A derived attribute birthDates, computed as shown in Listing 1.4, gathers the birthdates of all employees.

Listing 1.4. birthDates derived attribute [5, p. 29]

```
context Company::birthDates : Bag(Date)
  derive: self.employee->collect( person | person.birthDate )
```

Start and end of work contracts are handled by the operations hire(Person p, String title) and fire(Person p), both implemented as fUML activities. In particular in Fig. 3 we show the fUML activity diagram for fire. In the next section we will describe the fUML semantics by example. Then it will be clear that the fire activity selects the Job of the given employee among all Jobs in the current Company, and deletes it. Since a call to fire may change the list of employees, it will generally trigger the re-computation of both invariant and derived feature in Listing 1.3 and 1.4.

Figure 2 gives an overview of the relations between the languages we mentioned so far. The set of UML concepts (or metaclasses) can notably be divided into structural concepts (e.g., Class, Property, Operation, Interface), and behavioral ones (e.g., Activity, Action, StateMachine). Associations between structural and behavioral concepts enable jointly modeling both aspects. fUML considers only subsets of these structural and behavioral parts (e.g., excluding Interface, and StateMachine). Alf and OCL are two textual languages that are defined outside of UML, but that can be integrated in it using OpaqueBehavior and OpaqueExpression. Additionally, Alf specifies a transformation from its textual syntax to fUML elements, and a reverse transformation from fUML elements to Alf textual syntax. This enables usage of a textual syntax instead of relatively verbose diagrams.

A model of Company uses these languages for a precise executable specification of its semantics. Global analysis on Company could for instance detect that the fire activity does not impact the invariant in Listing 1.2. Optimization could avoid checking the invariant in this case. However such cross-language activities are not available in current executable modeling. In the following sections we introduce a translational approach, represented in Fig. 2 as the dashed arrow from OCL to fUML, that addresses this problem and others.

3 Integrating OCL in fUML by Translation

In this paper we propose a compilation strategy and a prototype compiler that takes as input an executable model, written by the joint use of fUML and OCL, and returns an OCL-free fUML model, where the OCL part has been translated into fUML behavior. To give a first view over our objective, this section illustrates the result of the compilation of the Company model from the previous section.

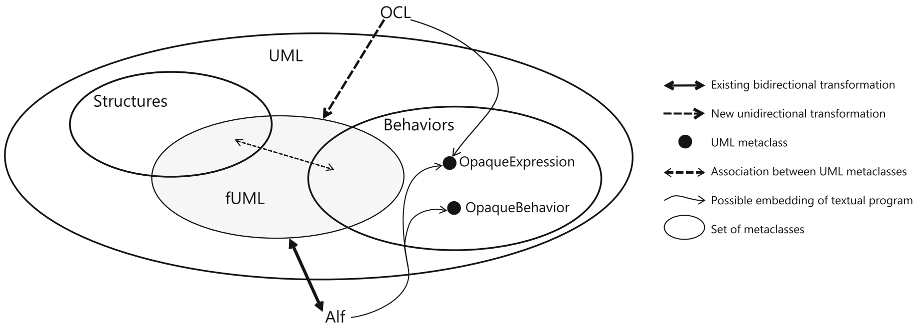

Fig. 2. Overview of relations between UML, fUML, Alf, and OCL

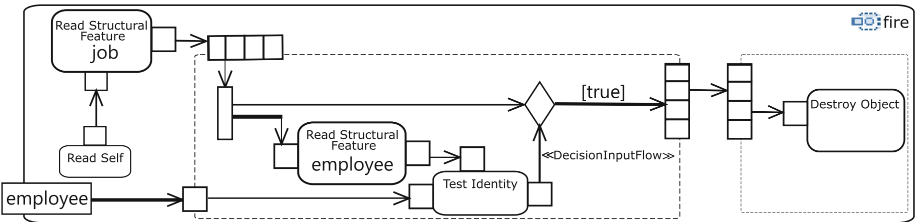

Fig. 3. The `fire` fUML activity

Bodies and pre/postconditions of OCL operations become side-effect free fUML activities. For instance the precondition of the `getCurrentSpouse()` OCL operation becomes the fUML activity in Fig. 4. The structure of an fUML activity is given by object flows that propagate object tokens from inputs to outputs and control-flow edges that force an execution order by passing control tokens. The activity in Fig. 4 makes exclusive use of object flows: the `result` of the activity comes from the comparison by a `TestIdentityAction` of (1) a constant value `true` provided by a `ValueSpecificationAction` and (2) the feature `isMarried` of the current object, obtained by a `ReadStructuralFeatureAction` over the output of a `ReadSelfAction`.

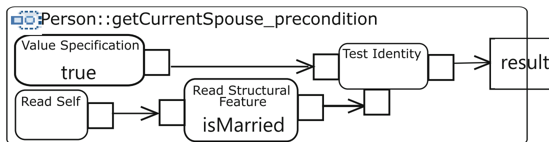

Fig. 4. `getCurrentSpouse` precondition activity (corresponding to Listing 1.1)

The body of `getCurrentSpouse()` becomes the activity in Fig. 5. Here the activity has to iterate through all the `marriages` to find the one that is not

`ended`. This is done in fUML by an `ExpansionRegion` (the dashed rectangle). An `ExpansionRegion` expects a collection as input (represented in the fUML semantics as a sequence of object tokens flowing on the link) and the content of the region is executed once for each element of the collection. The expansion node of the `ExpansionRegion` passes the elements of the input collection one by one through its outgoing object flow. The result of all iterations is gathered by the output node of the `ExpansionRegion`. In Fig. 5 each `marriage m` flows through a `fork` that passes it to (1) a flow that accesses the `ended` attribute and compares it to the constant `false`, and (2) a `DecisionNode` (the diamond) that will let the marriage pass through only if the result of the previous test is `true`. The output node of the expansion region will gather the expected result, i.e. the `marriage` that is not `ended`, and the `spouse` of this `marriage` will be returned to the caller.

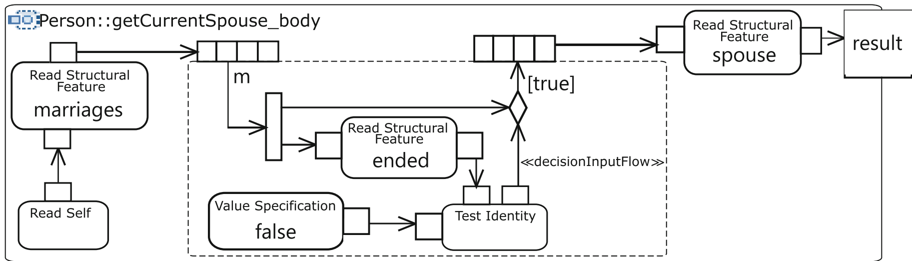

Fig. 5. `getCurrentSpouse` body activity (corresponding to the operation body in Listing 1.1)

The translation of the OCL expressions for precondition and body does not complete the translation of the `getCurrentSpouse()` operation to fUML. We still need to represent the implicit precondition semantics: the body is evaluated only if the precondition is satisfied, otherwise the behavior is undefined. We propose to describe this semantics in fUML by providing a wrapper for the operation, as shown in Fig. 6. The wrapper method starts by launching the evaluation of the precondition using a `CallOperationAction`. Depending on the result of the precondition, one of two possible `StructuredActivityNodes` (shown as dashed rectangles sub-activities) is executed: (1) if the precondition is true, the activity representing the operation body is called and its result returned, (2) if the condition is false, the missed precondition is reported (e.g., using a logger or the standard output through the `WriteLine` operation of the standard fUML library), and no result token is returned. An alternative (undefined) behavior would be to terminate the execution. However, fUML does not support exceptions, and does not provide means to terminate the execution engine. Note that this wrapping approach can also be used for post-conditions.

The translation of the invariants in Listing 1.2 (i.e., spouses are adults) and 1.3 (i.e., companies are big) are respectively shown in Figs. 7 and 8. The structure

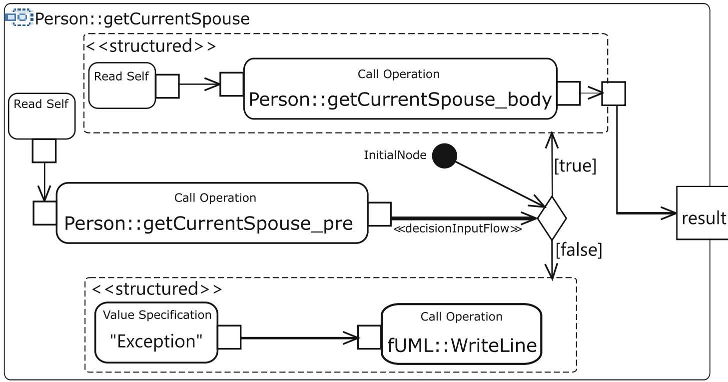

Fig. 6. getCurrentSpouse main activity (corresponding to Listing 1.1)

is straightforward, since for each operation of the OCL library (e.g., `notEmpty`, `implies`, `>=`, `and`, `size`) we reuse the equivalent operation in the fUML or Alf standard libraries. The fUML standard library mostly defines native functions for primitive types, and input-output. The Alf standard library extends it with higher-level functions, many of which are implemented in fUML.

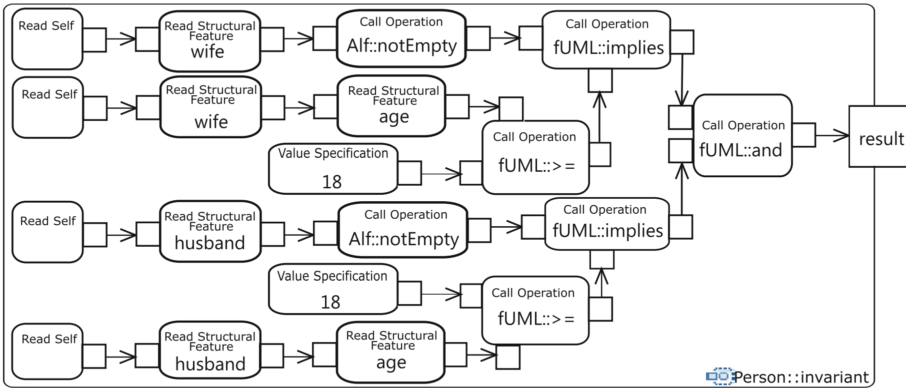

Fig. 7. Invariant on Person (corresponding to Listing 1.2)

Figure 9 is the result of the compilation of the derived feature in Listing 1.4 (`birthDates`). In this case we produce a getter operation for the derived feature, that performs the attribute (re-)computation. The OCL `collect` is compiled as an `ExpansionRegion` that iterates on `employees` and gathers their `birthDates`. As fUML does not support derived features, we propose to turn them into getters without backing fields. This also requires replacing every action accessing a derived property into a call to the corresponding getter. The approach works

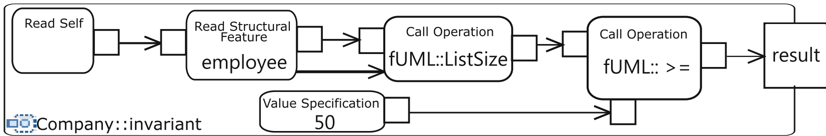

Fig. 8. Invariant on Company (corresponding to Listing 1.3)

in fUML because property values are not observable (while it could fail in UML where `ChangeEvents` can be leveraged to observe property values).

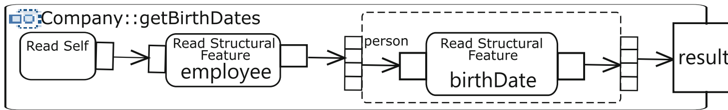

Fig. 9. getBirthDates activity (corresponding to Listing 1.4)

We have now considered the translations of all OCL examples from the previous section. Their interaction with regular fUML activities like `fire(p : Person)` from Fig. 3 can now be explained. Because it has side-effects, `fire(p : Person)` must be wrapped in a way similar to what was done for `getCurrentSpouse()`. This takes care of checking the invariant from Listing 1.3 after each change to the list of employees. The getter created from the `birthDates` derived feature does not need a special treatment, since it will return an updated value the next time it is accessed.

Finally, we quickly consider some uses of OCL in other parts of a UML model. It can notably be used everywhere a `ValueSpecification` can be used. We only have space to mention three possibilities here:

1. **Activity simplification.** A `ValueSpecificationAction` could actually contain an OCL expression instead of a constant literal. Figure 10 shows how the `fire(p : Person)` activity from Fig. 3 can be simplified by leveraging this possibility. The purely side-effect free part of the activity has been expressed as an OCL expression embedded in a `ValueSpecificationAction`. Given this new fUML+OCL activity, deriving the pure fUML version is simply a matter of compiling the OCL expression in the same way we compile bodies, pre- and post-conditions, invariants, and derive expressions.
2. **Default values of properties** can be specified in OCL with the `init` kind of expression, or with `Property.defaultValue` from the UML metamodel. They could be compiled into actions that initialize the corresponding properties from within constructors.
3. **Default values of parameters** can similarly be expressed in OCL. These OCL expressions could be compiled, and leveraged to generate alternative versions of a given operation, in which parameters with default values would disappear, and be computed instead.

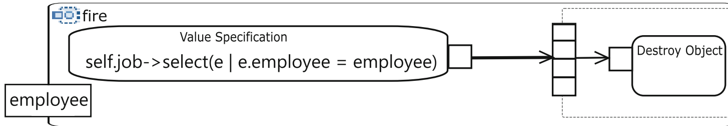

Fig. 10. A variant of the fire(p : Person) activity (Fig. 3) simplified using OCL

4 Compiling OCL to fUML

To obtain the result outlined in the previous section we defined a compilation scheme of OCL to fUML, that we describe in this section. The mapping is specified as an ATL transformation and we outline here its points of interest. We address three problems: mapping operations from the OCL standard library to the fUML or Alf standard libraries (Sect. 4.1), defining translation patterns for OCL expressions (Sect. 4.2), and translating the fUML extensions for embedding OCL code into pure fUML (Sect. 4.3).

4.1 Mapping the OCL Library to fUML

Table 1 summarizes the main correspondences between operations for OCL data types and fUML/Alf[3]. We can identify three kinds of correspondence:

- Operations on **OclType and OclAny** are directly translated into specific fUML actions, properly parametrized.
- Operations on **primitive data types** (`Boolean`, `String`, `Integer`, `Real`) translate into calls to primitive behaviors in the fUML standard library.
- Operations on **collections** translate into calls to behaviors in the Alf standard library. Note that this mapping does not include iterators. Alf does not include support for lambda expressions, so its library cannot include an implementation of general iterators. We therefore support iterators by statically compiling them to fUML patterns.

4.2 Translating OCL Expressions

In Figs. 11, 12, 13, and 14 we illustrate some examples of compilation patterns for OCL. These patterns are analogous to those given in the fUML specification [1, Appendix A] for Java to fUML translation.

The pattern in Fig. 11 is generated for all OCL conditional expressions. The compiled condition propagates its boolean result through an object flow to a `DecisionNode`. Depending on the boolean value, two control-flow edges

[3] The complete mapping for the OCL standard library is available at the project repository https://github.com/atlanmod/OCL2fUML.

Table 1. Mapping the OCL standard library to fUML

<table>
<tr><th>Context</th><th>OCL operation</th><th>Generated fUML</th></tr>
<tr><td>OclType</td><td>allInstances</td><td>ReadExtentAction</td></tr>
<tr><td>OclAny</td><td>=</td><td>TestIdentityAction</td></tr>
<tr><td></td><td><></td><td>TestIdentityAction</td></tr>
<tr><td></td><td></td><td>CallBehaviorAction(fUML::Not)</td></tr>
<tr><td></td><td>oclIsKindOf</td><td>ReadIsClassifiedObjectAction</td></tr>
<tr><td>Boolean</td><td>[and | or | xor | not | implies]</td><td>CallBehaviorAction(fUML::[And | Or | Xor | Not | Implies])</td></tr>
<tr><td>String</td><td>[concat | size | substring]</td><td>CallBehaviorAction(fUML::[Concat | Size | Substring])</td></tr>
<tr><td>Integer,Real</td><td>[+ | - | * | / |>|<|≥|≤]</td><td>CallBehaviorAction(fUML::[+ | − | ∗ | / |>|<|≥|≤])</td></tr>
<tr><td>Collection</td><td>size()</td><td>CallBehaviorAction(fUML::ListSize)</td></tr>
<tr><td></td><td>[includes | excludes | including | excluding | count | isEmpty | notEmpty | union | intersection | at | first | last | union | includesAll | excludesAll | insertAt]</td><td>CallBehaviorAction(Alf::[includes | excludes | including | excluding | count | isEmpty | notEmpty | union | intersection | at | first | last | union | includesAll | excludesAll | insertAt])</td></tr>
</table>

pass the control token (coming from an initial node) to the compiled expression from the `then` or `else` part. Thanks to the use of control flows the part that does not receive the control token is not evaluated at all. Finally a `MergeNode` gathers the result of the active part (`then` or `else`).

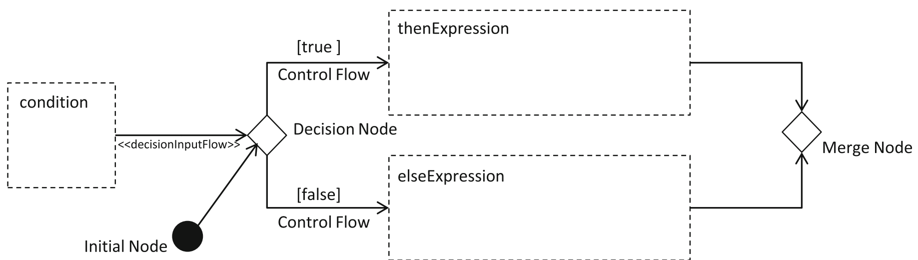

Fig. 11. Compilation pattern for OCL `if`

OCL iterators are generally compiled to fUML expansion regions, as we show in the following examples. For clearly understanding the correspondence a key point is remembering that collections in the fUML semantics are represented

as sequences of object tokens that flow consecutively over an object flow. The expansion region considers the tokens one by one, and for each token it may compute a result and propagate it to the object flow outgoing from its output node.

Compiling the `collect` iterator is straightforward, since the default behavior of fUML expansion regions corresponds to a functional map semantics. As shown in Fig. 12 from each `collect` we generate an expansion region that takes the input collection as a sequence of object tokens through the object flow coming from the source. The expansion region directly includes the compilation result from the body of the collect. We only add a `ForkNode` to simplify the distribution of the iterator in case it is used multiple times in the body.

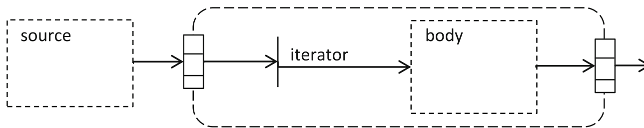

Fig. 12. Compilation pattern for OCL `collect`

The pattern in Fig. 13 represents the translation of an OCL `select` iterator. In this case the compiled body returns a boolean value that is used as decision input of a `DecisionNode`. If the boolean is `true` then the current value of the iterator is passed to the output node, otherwise it is discarded (because there is no `false` outgoing edge).

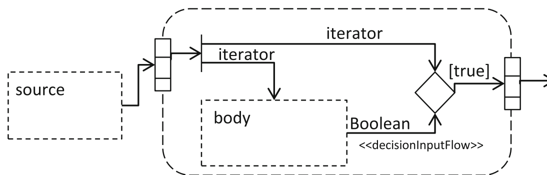

Fig. 13. Compilation pattern for OCL `select`

We compile an OCL `exists` iterator to the pattern in Fig. 14 that chains the `select` pattern of Fig. 13 with a `CallBehaviorAction` to the `notEmpty` behavior of the Alf library. In conformance to the OCL semantics the body of the `exists` is evaluated for the whole collection (the evaluation is not short-circuited when an element satisfying the body is found). When the entire collection has been evaluated a control flow launches the `notEmpty` behavior that returns the final result.

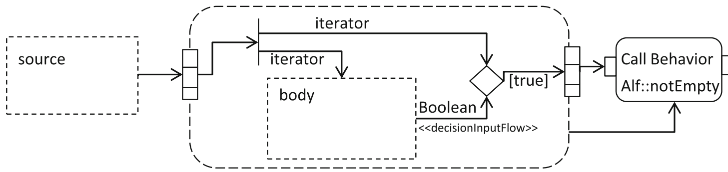

Fig. 14. Compilation pattern for OCL `exists`

4.3 Translating fUML Extensions

OCL expressions are included in our fUML models using mechanisms that are not part of the fUML specification, but are imported from UML. These mechanisms need to be translated into fUML elements to be executable. Depending on the role of the OCL expression, we generate a different fUML scaffolding:

- **body:** If the corresponding operation is missing from the fUML class model, we create a new operation. A wrapper activity with structure analogous to Fig. 6 is attached to the operation. We create also a separate private operation for the body, and we attach to it the activity generated from the OCL expression. The wrapper has a `CallOperationAction` towards the body operation.
- **pre, post, inv:** For each constraint we generate a new auxiliary operation associated with a side-effect free fUML activity that returns a boolean. Bodies of other operations in the model are wrapped in activities that check pre- and post-conditions of the operation and invariants of the class.
- **derive:** We create a getter operation (e.g., `get<FeatureName>`). We attach the fUML activity generated from the OCL expression to the getter. We replace all read accesses to the feature (`ReadStructuralFeature`) by calls (`OperationCallAction`) to the getter.
- **def:** We create a new operation and associated method.
- **init:** We add an `AddStructuralFeatureValueAction` to the class constructor to set the value of the property to the result of the compilation of the OCL expression.

5 Proof-of-Concept Implementation

In order to evaluate the feasibility of the approach presented in this paper, we partially implemented it. All OCL examples given in the previous sections have been translated into fUML using this implementation, and executed using an fUML execution engine. The present section describes this implementation.

We decided to use Moka[4] [2] as an fUML engine because (1) it is based on the reference implementation but (2) provides more tooling. Thanks to (1) we have confidence in its conformity to the fUML specification. And (2) means that activities like debugging are simpler than with the reference implementation.

[4] https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution.

Also, because Moka is integrated into Papyrus, we could use it to create the diagrams presented in this paper[5]. We used Moliz [6] in previous work [7] because we needed xMOF [8] (i.e., fUML lifted to the metametamodel). However, xMOF is not required for the present work, and showing that the approach can work with a full-featured UML editor like Papyrus seems more relevant.

The translation of OCL expressions into fUML is performed in two steps:

1. **Text-to-model parsing** yields a model representation of textual OCL expressions. We used the ATL parser for this purpose. This is possible because ATL expressions are written in a variant of OCL 2.0, which we call ATL-OCL. An additional shortcut consisted in writing OCL expressions in a separate file to simplify parsing. Parsing OCL expressions actually embedded in a UML model (in `OpaqueExpressions`) would not present significant issues.
2. **Model transformation** integrates the OCL model resulting from parsing into an fUML model. This transformation is written in ATL refining mode, which enables in-place updates. It takes as inputs both an extended fUML model, as well as an ATL-OCL model, and gives as output an updated version of the fUML model. The resulting fUML model contains the elements generated from the translation of all OCL expressions given as input. Additionally all fUML extensions have been integrated and removed.

Several other existing OCL parsers could have been reused. Notably, in an actual tool, leveraging Eclipse OCL would make more sense, because it closely follows the current OCL specification. The choice of ATL-OCL was motivated by the two following considerations: (1) familiarity with ATL-OCL enabled reduced development time (we have already worked on several ATL transformations that process ATL-OCL [9]); (2) reusability of ATL code helps amortize development cost. We already had an ATL-OCL to fUML transformation for a small subset as part of previous work on an ATL to fUML transformation [7]. Furthermore, we also intend to integrate the new transformation (extended to a larger subset of OCL) into the ATL to fUML transformation.

For the OCL to fUML model tranformation, we used only declarative ATL, which means that the rules are very close to a mapping specification. Each rule translates an OCL metaclass into a set of fUML elements. To connect the different fUML elements that are generated by different rules, we use a solution that can be summarized in two steps. The first step creates from each OCL expression, a set of fUML elements that should include one *result* element and may contain a set of *input* elements. The *result* and *input* elements can be represented by the input node, output nodes, the fork nodes, the join nodes, the input expansion region or output expansion region according to the corresponding source element. The second step aims to generate for each *input* element an object flow coming from the appropriate *result* element, e.g. the one generated from the parent OCL expression.

[5] The diagrams have nonetheless been modified manually in Inkscape (https://inkscape.org/), mostly to improve whitespace usage.

Our compiler supports most OCL constructs with the following notable exceptions: (1) tuples and iterate expressions could be added with relatively little work, (2) introspection of previous values of object properties, as well as of all messages sent by an operation in post-conditions would require detailed bookkeeping, and is likely to have a significant overhead, (3) `oclIsInState` is not relevant because fUML does not include state machines yet, at least until PSSM is standardized.

6 Discussion

A tool based on the OCL to fUML approach has two main *usage scenarios*:

1. *Synchronization* from OCL to fUML activities could be used to simplify the definition of fUML activities. Users would write a couple of OCL expressions, then run the transformation to translate them into fUML. Then, they would continue working on the resulting model, and iterate the process. This is similar to what can be done with Alf, except that Alf can also go back from fUML activities into code.
2. *As a pre-processor* for an fUML execution engine. Users would express significant parts of their model in OCL. They would only use the transformation to fUML before loading their model into an fUML engine (e.g., for simulation), but would never modify the resulting model. Then, they would continue working on the original model, and iterate the process.

A combination of these two scenarios is also possible. Parts of the OCL expressions may be used to specify fUML activities, and therefore be translated as soon as possible into fUML. Other parts could be left as OCL expressions (e.g., preconditions, invariants), and only translated to fUML before execution.

The synchronization scenario is likely to require more complex tooling. Whereas a pre-processor may be invoked transparently (or as a command line tool), synchronization between code and fUML is non trivial and requires a well-designed user interface, as the integration of Alf into Papyrus shows [10]. Additionally, later modifications may be more difficult. For instance, removing an OCL precondition simply consists of removing the link between operation and constraint. However, removing a precondition compiled to fUML requires editing an activity.

We observe that the OCL to fUML transformation is similar to the Alf to fUML transformation (see Fig. 2). Therefore, users could use *OCL in place of Alf* to textually specify some fUML activities: those that are side-effect free. This is even possible on UML models that contain more than just fUML and the relatively simple extensions compiled in our work, although fUML execution may not be possible. Some users may prefer OCL because it is a purely functional language. For instance, it is not possible to inadvertently insert side-effects in query operations written in OCL. However, the missing fUML to OCL reverse transformation would make OCL less convenient than Alf for this purpose. Moreover,

this missing transformation would necessarily be partial because fUML activity with side-effects would not be translatable into OCL expressions.

Since OCL has been in use for quite some time (OCL 1.1 published in 1997), some users are likely to know it much better than Alf (version 1.01 published in 2013), and fUML (version 1.0 published in 2011). Being able to translate OCL expressions into fUML could help them getting started with fUML. Also, even if they do not write OCL expression in order to generate fUML activities, playing with an OCL to fUML compiler could be a significant learning aid.

Our integration solution is based on a compilation approach: OCL expressions are translated into fUML activities. An alternative approach would be to evaluate OCL expressions using an *interpreter written in fUML*. Although both approaches have pros and cons, we preferred the compilation one because it avoids double interpretation of OCL (i.e., interpretation of OCL by an fUML interpreter itself interpreted by an fUML engine). Current implementations of fUML (the reference implementation, as well as Moka and Moliz, which are based on it) are all interpreters. Another possibility would be to extend the fUML interpreter with support for OCL interpretation. This could be done by coupling it with an existing black-box OCL interpreter, but then one would lose the fine grained control over execution provided by the fUML interpreter.

While working on the translation presented here, we considered the following *possible extensions to fUML*:

1. *Integration with State Machines.* Ongoing work at OMG on a state machine extension to fUML could make use of our OCL to fUML translation approach. Transition guards can notably be specified in OCL.
2. *Built-in support for pre- and post-conditions.* The previous sections presented one possible integration of pre- and post-conditions into fUML. However, other integrations are possible, notably with respect to when they are evaluated. One possible way to support several integration strategies would be to add options to the compiler. Another way would be to extend fUML with a strategy similar to the existing `DispatchStrategy`, and `ChoiceStrategy` (among others), which respectively enable plugging alternative operation dispatch algorithms, and alternative algorithms to select which action to perform next among those activated.
3. *Anonymous Functions.* Translation of OCL iterator expressions such as *select* and *collect* does not leverage a generic library but rather always generates expansion regions. If fUML supported anonymous functions, using a library approach could be quite simple. Alternatively, we could have used classes as function wrappers, like in Java, or even implemented defunctionalization.
4. *Exceptions.* Violations to pre-, post-conditions and invariants encoded in fUML (Sect. 3), cannot result in fail-fast terminations, but rather in undefined behaviors. Exceptions are one UML feature which would be particularly useful to integrate in fUML for this purpose. An alternative would be to add a native function to the fUML execution engine that would terminate its execution (similarly to what `System.exit(1)` does in Java).

7 Related Work

Approaches like [11] focus on UML model validation with respect to OCL pre- and post-conditions, as well as invariants. In contrast, our approach focuses on model executability. For instance, we also integrate OCL in places where its execution is part of the execution of the model (e.g., operation bodies, derived features), not just to validate it. However, our integration of the pre- and post-conditions is weaker, and closer to what can be achieved with asserts in Java.

In [12], the authors extend OCL with the capability to perform side effects (e.g., assign values to properties, create instances). They then use this extended OCL as an action language for an executable UML. To support UML actions, they further define a mapping from actions to extended OCL expressions. This is basically the opposite of the mapping we propose from OCL to UML actions.

Several UML editors, like Papyrus, provide execution of OCL. However, this execution is typically supported using a dedicated interpreter. Tools that also integrate fUML do not necessarily provide the capability to call OCL from fUML. When they do they typically exhibit a problem mentioned in Sect. 6: the OCL and fUML interpreters are only loosely connected.

The Alf specification can also be considered as a related work, but its relation to our work has already been described in Sect. 6. Another aspect of our approach is that it enables evaluation of OCL expressions in tools that do not support OCL but only fUML. We then see fUML as a kind of virtual machine. This aspect and corresponding related work is described in our papers [7,13].

8 Conclusion

In this work, we have presented a compiler to generate a fully executable fUML model from a user-specified model containing both fUML activities and OCL constraints. We discussed the practical advantages of building executable models using both languages, and having them executed at the same level of abstraction. In previous work we already used the analogy of fUML as a modeling virtual machine, a common platform for execution of modeling tools. This paper is a step in this path, by showing the integration of constraint checkers and query languages in fUML. Since OCL is integrated as expression language in several other MDE tools, this work may be leveraged to port them to fUML as well.

References

1. Object Management Group (OMG): Semantics of a Foundational Subset for Executable UML Models (fUML), v1.2.1, January 2016
2. Guermazi, S., Tatibouet, J., Cuccuru, A., Seidewitz, E., Dhouib, S., Gérard, S.: Executable modeling with fUML and alf in papyrus: tooling and experiments. In: 1st International Workshop on Executable Modeling, pp. 3–8 (2015)
3. Object Management Group (OMG): Precise Semantics Of UML Composite Structures (PSCS), v1.0. http://www.omg.org/spec/PSCS/1.0/, October 2015

4. Object Management Group (OMG): Concrete Syntax For A UML Action Language: Action Language For Foundational UML (ALF), v1.0.1. http://www.omg.org/spec/ALF/1.0.1/, October 2013
5. Object Management Group (OMG): Object Constraint Language (OCL), v2.4. http://www.omg.org/spec/OCL/2.4/, February 2014
6. Mayerhofer, T., Langer, P.: Moliz: a model execution framework for UML models. In: Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards. MW 2012, pp. 3: 1–3: 2. ACM, New York (2012)
7. Tisi, M., Jouault, F., Delatour, J., Saidi, Z., Choura, H.: fUML as an assembly language for model transformation. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 171–190. Springer, Heidelberg (2014)
8. Mayerhofer, T., Langer, P., Wimmer, M., Kappel, G.: xMOF: executable DSMLs based on fUML. In: Erwig, M., Paige, R.F., Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 56–75. Springer, Heidelberg (2013)
9. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 18–33. springer, Heidelberg (2009)
10. Seidewitz, E., Tatibouet, J.: Tool Paper: Combining Alf and UML in Modeling Tools - An Example with Papyrus. In: Brucker, A.D., Egea, M., Gogolla, M., Tuong, F. (eds.) OCL@MoDELS. vol. 1512 of CEUR Workshop Proceedings, CEUR-WS.org, pp. 105–119 (2015)
11. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. **69**(1), 27–34 (2007)
12. Jiang, K., Zhang, L., Miyake, S.: Using OCL in executable UML. In: Proceedings of the Workshop Ocl4All: Modeling Systems with OCL at MoDELS 2007. vol. 9, Electronic Communications of the EASST (2008)
13. Tisi, M., Jouault, F., Delatour, J., Saidi, Z., Choura, H.: fUML as an assembly language for model transformation. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 171–190. Springer, Heidelberg (2014)