Model-Driven Development of an Interpreter for the Object Constraint Language

Gonzalo Sintas, Leticia Vaz Lutz, Daniel Calegari, Marcos Viera
Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, 11300
{gonzalo.sintas,leticia.vaz.lutz,dcalegar,mviera}@fing.edu.uy

***Abstract*—The Object Constraint Language (OCL) plays a central role in many Model-Driven Engineering (MDE) initiatives for expressing invariant conditions in models, among other uses. It is defined as a side-effect free language combining model-oriented and functional features. There are works that explore the use of the functional paradigm for the interpretation of OCL such that most of the functional infrastructure could be predefined and automatically generated. In this work we present the MDE-based automatic generation of such functional infrastructure based on Haskell. We also present how the infrastructure is connected with the Eclipse Modeling Framework for final users, allowing the internal complexity to be transparent to them. We provide a different perspective on the interpretation of OCL, together with a sandbox that allows to evaluate in a unified way several proposals that already exist in the scientific community, as well as to collaborate with the migration of functional aspects to the MDE paradigm.**

Keywords: Object Constraint Language, Model-Driven Engineering, Haskell, Eclipse Modeling Framework.

I. Introduction

The Model-Driven Engineering paradigm (MDE, [1]) adopts a model-centric approach for the software engineering process (construction, maintenance, reverse engineering, etc.) Models are abstractions of a system (or some aspects of it) allowing us to deal with its intrinsic complexity in a simplified manner. They are formally defined based on metamodels, which capture the syntax and semantics of a modeling language and thus provide the context needed for expressing well-formed constraints for them (known as the conformance relation). Moreover, the use of automated mechanisms (model transformations), e.g.: for building the software system, tends to improve efficiency on the whole engineering process.

In this context, the Object Constraint Language (OCL, [2], [3]) plays a central role. In many concrete MDE initiatives, e.g.: the Model-Driven Architecture (MDA, [4]) approach, the OCL is used to specify conditions (invariants) that cannot be captured by the structural rules of the metamodeling language, and also used for constraining and computing object values in the definition of model transformation rules. OCL is also used for the description of pre- and post-conditions on operations, and the specification of guards in behavioral diagrams, among many other purposes.

The OCL is defined as a side-effect-free language combining model-oriented and functional features, e.g. type inheritance and functions composition, respectively. In the last few years, many authors propose the inclusion of functional features in the language, e.g. pattern matching [5], lambda expressions [6] and lazy evaluation [7]. These concepts have a direct representation in functional programming languages (e.g.: Haskell [8]) and could be not easily interpreted following a model-oriented approach. Thus, a functional approach comes as a reasonable alternative for exploiting such features.

To cope with this kind of situations, a separation of duties between software developers is usually proposed, giving rise to different technological spaces [9], i.e.: working contexts with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. In this case, on the one side, MDE experts define models and transformations and, on the other, functional experts conduct an evaluation or verification process. The connection between the technological spaces is often aided by some (semi)automatic circular process which translates the MDE elements to their functional representation, and also retrieves some feedback to the MDE experts.

In previous work [10] we have explored a functional approach to support the construction of an OCL interpreter with respect to its use for expressing invariant conditions in models. We have presented a Haskell-based representation of the main aspects of OCL and we have discussed how it provides a direct and clear interpretation for advanced OCL features proposed in the literature. We claimed that the functional infrastructure can be predefined and automatically generated, but it required further work in order to be put into real action.

In this work we continue reducing the gap between both technological spaces by providing an MDE-based automatic generation of such functional infrastructure. In particular, we develop a predefined functional version of the OCL library, and we provide an automatic transformation of models, metamodels and OCL invariants into Haskell. We also present Haskell OCL [11], an Eclipse Modeling Framework (EMF, [12])-based tool for final users, allowing to both: generating the infrastructure and also checking invariants, in such a way that the internal complexity is transparent to them. These results provide a different perspective on the interpretation of OCL, together with a sandbox that allows to evaluate in a unified way several proposals that already exist in the scientific community, as well as to collaborate with the migration of functional aspects to the MDE paradigm.

We believe that the main ideas and solutions we propose can

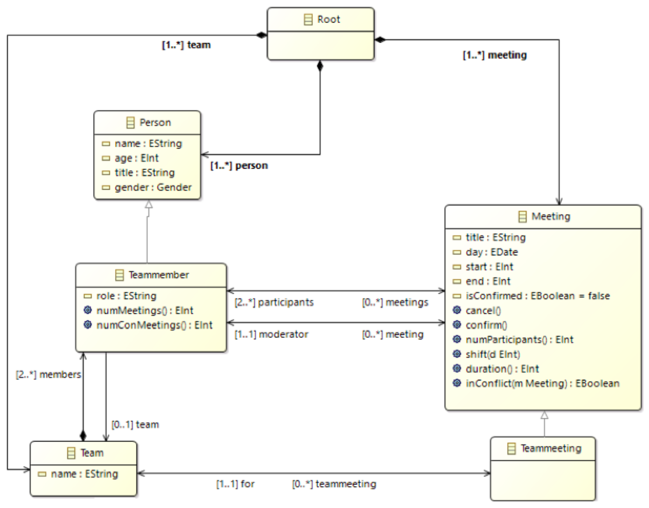


Figure 1: Team Meeting (Metamodel)

be understood by non-functional experts. However, in order to reach a wider audience, as far as possible we hide the details of the functional representation. In order to fully understand them, we suggest to refer to [10].

The rest of this paper is organized as follows. In Section II we define a case study used for the general understanding of the ideas provided in this paper. In Section III we present the general approach for the functional interpretation of OCL invariants. In Section IV we describe the automatic generation of the functional infrastructure. In Section V we provide details on how the infrastructure is connected with the EMF. In Section VI we present related work and provide some discussions about compliance of our proposal with the OCL. Finally, in Section VII we present some conclusions and an outline of future work.

II. Case Study

All along this paper, we consider the following case study as a proof of concepts and for a better understanding of the technical details. Complete information on the source code of our running example is available at [11]. The Team Meeting example was taken from [13] and adapted to our purposes. Besides it is simple, it involves a very expressive subset of OCL.

The metamodel in Figure 1 represents teams of people and team meetings, such that each meeting has certain number of participants and a moderator from the same team.

In Figure 2 there are some of the OCL invariants that must be ensured for any model. The invariants consider primitive and user-defined types, operations on collections, navigation through properties, and comparison operators.

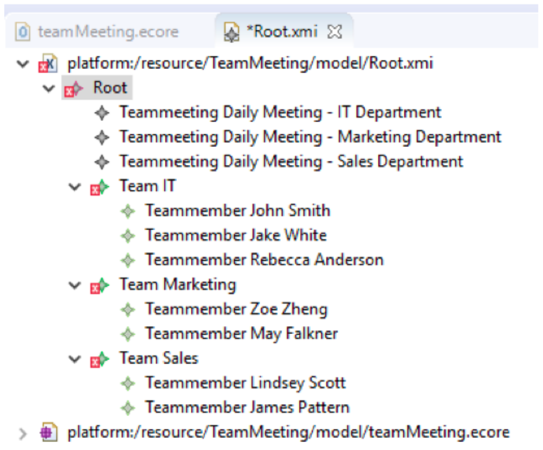
In Figure 3 there is a model satisfying the constraints, in which three meetings are defined, one for each team (IT, Marketing and Sales), such that every member of the team participates in its corresponding meeting.

```
— Inv1: A meeting ends after it starts
context Meeting inv:
  self.end > self.start

— Inv2: A meeting has at least 2 participans
context Meeting inv:
  self.participants -> size() >= 2

— Inv3: A teammeeting has to be organized for a whole team
context Teammeeting inv:
  self.participants -> forAll(team = self.for)
```

Figure 2: Team Meeting (OCL invariants)


teamMeeting.ecore   *Root.xmi

platform:/resource/TeamMeeting/model/Root.xmi
  Root
    Teammeeting Daily Meeting - IT Department
    Teammeeting Daily Meeting - Marketing Department
    Teammeeting Daily Meeting - Sales Department
    Team IT
      Teammember John Smith
      Teammember Jake White
      Teammember Rebecca Anderson
    Team Marketing
      Teammember Zoe Zheng
      Teammember May Falkner
    Team Sales
      Teammember Lindsey Scott
      Teammember James Pattern
platform:/resource/TeamMeeting/model/teamMeeting.ecore

Figure 3: Team Meeting (Model)

III. Functional Interpretation of OCL

In Figure 4 we depicted the functional interpretation of OCL as a process starting with the definition of models, metamodels and OCL invariants, going through the generation of the functional infrastructure and the checking of invariants, and ending with the visualization of the results. Our Haskell OCL tool is defined as an EMF plugin in which three different tools collaborate: Eclipse OCL[1], Acceleo[2] and Haskell. In what follows we present a general overview of the activities to be performed during the process, their inputs, outputs and tool support.

A. Model domain and invariants

The first activity to be performed is the modeling of the different domain elements, i.e.: a metamodel together with their corresponding OCL invariants, and a model in which the invariants must be verified in order to check its conformance with the metamodel. Eclipse OCL is used for this purpose, since it is an EMF-based plugin providing modeling capabilities for models, metamodels and OCL invariants, as well as an OCL interpretation engine. The metamodel, as the one in Figure 1, is defined using Ecore; the OCL invariants, as those in Figure 2 are embedded within the Ecore metamodel,

[1]Eclipse OCL: https://projects.eclipse.org/projects/modeling.mdt.ocl
[2]Acceleo: http://www.eclipse.org/acceleo/

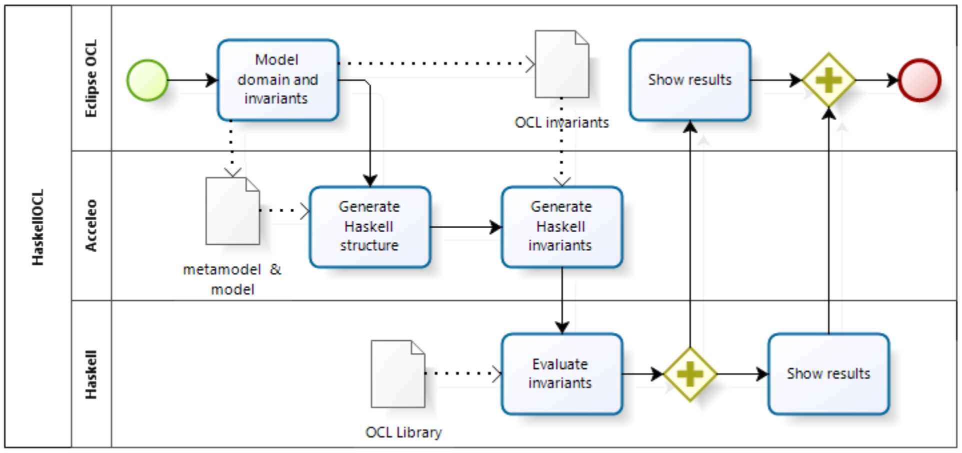


Figure 4: HaskellOCL process

and the models, as the one in Figure 3, are well-formed XMI instances of the Ecore metamodel. From a practical point of view, by using Eclipse OCL we can assume that parsing and type checking issues are satisfied by the environment.

B. Generate Haskell structure & Haskell invariants

The OCL infrastructure is generated in two steps (as will be detailed in Section IV). We use Acceleo, an EMF-based implementation of the MOF Model to Text Language (MTL) standard, providing model transformation capabilities. The first step involves taking the metamodel and model provided by the first activity and generating the functional structure (in Haskell) that will be used for expressing the OCL constraints. The transformation is based on the ideas already presented in [10]. The second step takes as an input the OCL invariants defined in the first activity and generates the Haskell-based representation of the invariants, been consistent with the functional structure already generated.

The output is a Haskell file (`.hs`) with the complete functional infrastructure. Although the functional infrastructure could be not easily readable for an inexperienced user, it is automatically generated and there is a direct representation of the OCL invariants mimicking its structure. Besides the purpose of this output is to be used for transparently running the evaluation (the transformation also generates a main function for running the evaluation), it is made available in order to allow functional experts to directly experiment with it.

C. Evaluate invariants & Show results

Haskell provides the environment for the functional interpretation of OCL constraints. Once the Haskell file is generated, the tool runs the evaluation of each OCL invariant within the file with respect to the given model. The execution needs a predefined functional OCL library. This library does not depend on the concrete information provided by the metamodel and models, so it is statically defined and not generated on each use. In Section VI we summarize the current level of compliance of this library with respect to the OCL standard.

The Haskell evaluation allows a clean handling of errors and a precise definition of the four-valued semantics of OCL (True, False, Invalid or Null). In this sense, Haskell provides the result of the evaluation as a list of values, one for each invariant. These results can be seen in the command console. However, in parallel, this information is also displayed in the Validity View of Eclipse OCL. The Validity View provides a much more detailed view of the problems by relating the invariants with their context, i.e.: the main metamodel element for which the invariant is defined. In Section V we show how this is used.

IV. Generation of the Functional Infrastructure

We defined Acceleo model to text transformations to generate a Haskell file out of an Ecore model and its corresponding metamodel and invariants. The transformations are based on the representation introduced in [10].

A. The OCL Library

In our previous work we introduced a library that defines a type for the invariants and a set of functions to construct and execute them with. An invariant has type `OCL m (Val` **`Bool`**`)`, which can be read as an OCL expression that applies to a model represented by a value of type `m` and returns a boolean. The `Val` type that wraps the boolean represents the OCL four-valued logic with the notion of truth, undefinedness and nullity.

We defined operators to represent the object navigation (|.|) and the collection navigation (|−>|). We also provided a somehow limited support to operate over booleans, integers, reals and collections into the four-valued logic.

In this work we extended the library in order to increase the level of compliance with the OCL standard. In Table I there is a summary of the compliance with the OCL (as defined in [2]), taking the original work and the last version of the Eclipse OCL tool[3] as references. We use a semaphore-like notation where: green, yellow and red means that the OCL aspect

[3]Eclipse Oxygen OCL 6.3 Compliance: https://wiki.eclipse.org/OCL/Compliance

Table I: OCL compliance and limitations

	Original work [10]	**Haskell OCL**	**Eclipse OCL**
OCL Constructs			
`context` Specifies the context for OCL expressions	✓	✓	✓
`inv` States a condition that must always be met by all instances of a context type	✓	✓	✓
`pre/post` States a condition that must be true at the moment when an operation starts/end its execution	✗	✗	~ [j]
`init` Specifies the initial value of an attribute or association role	✗	✓	✓
`derive` Specifies the value of a derived attribute or association role	✗	✗	✓
`body` Defines the result of a query operation	✗ [a]	✗ [a]	✓
`def` Introduces a new attribute or query operation	✗	✓	✓
`package` Specifies explicitly the package in which OCL expressions belong	✗ [b]	✗ [b]	✓
OCL Expressions			
`self` Denotes the contextual instance	✓	✓	✓
`result` In a postcondition, denotes the result of an operation	✗	✗ [a]	~ [j]
`@pre` In a postcondition, denotes the value of a property at the start of an operation	✗ [a]	✗ [a]	~ [j]
Navigation Navigation through attributes, association ends, association classes, and qualified associations	~ [c]	~ [c]	✓
`if-then-else` expression Conditional expression with a condition and two expressions	✓	✓	✓
`let-in` expression Expression with local variables	✗	✓	✓
Messaging Indicates that communication has taken place	✗ [d]	✗ [d]	~ [j]
OCL Standard Library			
`Boolean` Type Values: `true` / `false`	✓	✓	✓
`Boolean` Operations Operations: `or`, `and`, `xor`, `not`, =, <>, `implies`	~ [i]	✓	✓
`Integer/Real` Type Values: -10, 0, 10, ..., -1.5, 3.14, ...	~ [i]	✓	✓
`Integer/Real` Operations Operations: $=$, $<>$, $<$, $>$, $+$, $-$, $*$, $/$, `mod`, `div`, `max`, `round`, ...	~ [i]	✓	✓
`UnlimitedNatural` Type Values: $*$	✗	✗	✓
`String` Type Values: `"value"`	✗	✓	✓
`String` Operations Operations: $=$, $<>$, `concat`, `size`, `toLower`, `substring`, ...	✗	✓	✓
`OCLAny` Supertype of all UML and OCL types	✗	✗ [e]	✓
`OCLAny` Operations Operations defined for any type: $=$, $<>$, `oclIsNew`, `oclAsType`, `T::allInstances`, ...	~ [i]	~ [f]	~ [k]
`OCLVoid` Type with one single instance (`undefined`) that conforms to all others types	~ [g]	~ [g]	✓
`OCLMessage` Messages that can be sent to and received by objects	✗ [d]	✗ [d]	✗
`Tuple` Type A tuple consists of named parts each of which can have a distinct type	✗	✓	✓
`Collection` Types Four collection types: `Set`, `OrderedSet`, `Bag`, and `Sequence`	~ [i]	✓	✓
Collection operations Operations: `any`, `append`, `asBag`, `count`, `collect`, `excludes`, `exists`, `first`, ...	~ [i]	~ [h]	✓

[a] Operations within metamodels, and pre-/post-conditions on operations, are not supported.
[b] Syntactic sugar, can be easily supported.
[c] Association classes and qualified associations are not supported in our metamodels.
[d] Messaging is for other OCL uses, not for invariants definition.
[e] `OCLAny` is not defined as a type, but its operations are implemented for any type.
[f] The following operations are not supported: `oclIsInState`, `oclIsNew`, `oclType`.
[g] `OCLVoid` is not defined as a type, but `undefined` considerd as a possible value during evaluation.
[h] The following operations are not supported: `flatten`, `sortedBy`, `collectNested`.
[i] We support a very limited version in comparisson with the one defined for this work.
[j] Supports parsing of the expression, but not its interpretation.
[k] The following operations are not supported: `oclIsInState`, `oclIsNew`.

is fully, partially or not supported, respectively. We added the interpretation of new OCL constructs and expressions (e.g.: `def` and `if-then-else`, respectively), and we also provide full support for all the primitive types (except for `UnlimitedNatural`) and collection types, together with their corresponding operations. For this, we basically support our implementation on standard Haskell types and libraries, such that the functional representation is as direct as possible. In Section VI we provide deep insights in this sense, and in [11] there is the complete definition of the library.

B. Metamodels and Models

Classes, attributes, relationships and invariants of the metamodel, as well as the entities that are part of the model, are transformed to their corresponding representations in Haskell.

We start by translating every class of the metamodel to a Haskell datatype using the transformation of Figure 5. The following code shows the Haskell datatypes generated for Person and Teammember:

```
data Person = Person String Int String String (Maybe PersonChild)

data PersonChild = TeammemberCh Teammember

data Teammember = Teammember String [Int] [Int] [Int]
```

Without delving into details, the transformation generates for each class (`EClass`) a **`data`** named with the name of the class and one constructor with the same name. The fields of the constructor represent the properties of the class. In case of `EAttribute`, the type of the attribute is translated to the corresponding Haskell type; e.g. for the attribute `name` of `Person`, which has type `EString`, a **`String`** parameter is added to the constructor `Person`. For a `EReference` with multiplicity one, an **`Int`** parameter is added to the constructor, representing the identifier of the refereced object. In case of other multiplicities a `[Int]` (list of integers) is generated instead. This is the case for the refereces to `Meeting` and `Team` in `Teammember`. If the class has subclasses a field of type `ClassChild` is added, with `Class` the name of the class. This datatype is also generated (second part of the transformation of Figure 5), which defines one constructor (`ClassCh`) for each subclass (`Class`) with its corresponding type. For non abstract classes the `ClassChild` field is wrapped with a **`Maybe`** type, whose values are of the form (**`Just`** `v`), with `v` a value of type `ClassChild` or **`Nothing`**.

Inspired by the Zipper[14] structure, we allow to navigate up and down the class hierarchy by pairing the representation of the classes with their inmediate superclasses. This is done by the transformation of Figure 6, which for the examples of Person and Teammember generates:

```
type Person_ = (Person, ModelElement_)

type Teammember_ = (Teammember, Person_)
```

Some other transformations define some boilerplate that is useful to be able to navigate through the representation of the model in a uniform way. An example of this code is the following function to access to the attribute name from an element which is of class Person or any of its subclasses.

```
name :: Cast Model Person_ a
     => Val a -> OCL Model (Val String)
name a = upCast _Person a >>=
         pureOCL( \ (Person x _ _ _ _ , _) ->
                  return (Val x))
```

Again, without delving into the details of this code, the function goes upwards (`upCast`) in the hierarchy until it finds a value of type `Person`, and then returns the value representing the name.

Another example is a function to access to the moderator of a Meeting. In this case the value stored in the `Meeting` is an identifier, so the value of type `Teammember` has to be looked up into the model.

```
moderator :: Cast Model Meeting_ a
          => Val a -> OCL Model (Val Teammember_)
moderator a = upCast _Meeting a >>=
              pureOCL( \ (Meeting _ _ _ _ _ _ x _ , _) ->
                       lookupM _Teammember x)
```

In the transformation we define an abstract class ModelElement which is superclass of all the orphan classes of the metamodel. This class includes an integer which is used as the unique identifier the other elements can refer to.

```
data    ModelElement     = ModelElement Int ModelElementChild

data    ModelElementChild = PersonCh  Person
                          | TeamCh    Team
                          | MeetingCh Meeting
                          | RootCh    Root
```

Then, the Haskell representation of the model is a list of values of type `ModelElement`.

```
data    Model    = Model [ModelElement]
```

The model is generated by another Acceleo transformation, which takes as input an `EObject`. Figure 7 shows an example of a model generated for the Case Study.

C. OCL Invariants

The invariants are translated to Haskell functions. Since the Haskell representation mimicks the structure of the OCL invariants, the transformation basically travers the model representing an OCL invariant and produces for each subexpression, the corresponding string, also considering the context (the types involved) of the subexpression been transformed in order to use the corresponding Haskell functions. As an example, the following are the functions generated from the invariants of Figure 2. As can be seen, the functions are very similar to the original ones.

```
invariant1 = context _Meeting [inv1]
inv1 self = ocl self |.| end |>| ocl self |.| start

invariant2 = context _Person [inv2]
inv2 self = ocl self |.| participants
                                |->| size |>=| (oclInt 2)

invariant3 = context _TeamMeeting [inv3]
inv3 self = ocl self |.| participants |->|
     forAll (\a -> ocl a |.| team |==| ocl self |.| for)
```

```
[template public generateData(aClass : EClass)]
[comment @main /]
[file ('ACCELEO.hs', true)]

data [aClass.name/] = [aClass.name/] [for (attr : EAttribute | aClass.getAttributes())] [toHaskellType(attr.eType.name)/] [/for]
                                     [for (ref : EReference | aClass.getReferences())][if (ref.lowerBound=1 and ref.upperBound=1)]
                                                                                      ['Int'/][else]['[Int]'/][/if] [/for]
                                     [let allClasses : Sequence(EClass) = aClass.ancestors().eAllContents("EClass")]
                                                       [if (aClass.hasChildren(allClasses))]([aClass.name/]Child) [/if][/let]


[let allClasses : Sequence(EClass) = aClass.ancestors().eAllContents("EClass")]
[if (aClass.hasChildren(allClasses))]
data   [aClass.name/]Child = [for (elem : EClass | aClass.getDirectChildren(allClasses))]
                                     [if (i > 1)] | [/if][elem.name/]Ch ([elem.name/])[/for]
[/if][/let]

[/file]
[/template]
```

Figure 5: Class to Datatype Transformation (excerpt)

```
[template public generateZipper(aClass : EClass)]
[comment @main /]
[file ('ACCELEO.hs', true)]
type [aClass.name/]_ = ([aClass.name/], [if (aClass.hasFather())][aClass.getFather().name/][else]ModelElement[/if]_)
[/file]
[/template]
```

Figure 6: Class to Datatype Transformation

```
example = Model
        [ (ModelElement 0 (RootCh (Root [7,8,9,10,11,12,13] [1, 2, 3] [4, 5, 6] )))
        , (ModelElement 1 (MeetingCh (Meeting "Daily␣Meeting␣—␣IT␣Dept" "19/05/2018" 10 11 True [7,8,9] 7 []
                                       (Just (TeammeetingCh (Teammeeting 4))))))
        , (ModelElement 2 (MeetingCh (Meeting "Daily␣Meeting␣—␣Marketing␣Dept" "20/05/2018" 15 16 True [10,11] 10 []
                                       (Just (TeammeetingCh (Teammeeting 5))))))
        , (ModelElement 3 (MeetingCh (Meeting "Daily␣Meeting␣—␣Sales␣Dept" "19/05/2018" 8 9 False [12,13] 12 []
                                       (Just (TeammeetingCh (Teammeeting 6))))))
        , (ModelElement 4 (TeamCh (Team "IT" [7, 8, 9] [1] )))
        , (ModelElement 5 (TeamCh (Team "Marketing" [10, 11] [2] )))
        , (ModelElement 6 (TeamCh (Team "Sales" [12, 13] [3] )))
        , (ModelElement 7 (PersonCh (Person "John␣Smith" 53 "Chief" "Male" (Just (TeammemberCh (Teammember "Chief" [4] [1] ))))))
        , (ModelElement 8 (PersonCh (Person "Jake␣White" 26 "Developer" "Male" (Just (TeammemberCh (Teammember "Developer" [4] [1] ))))))
        , (ModelElement 9 (PersonCh (Person "Rebecca␣Anderson" 23 "Developer" "Female" (Just (TeammemberCh (Teammember "Developer" [4] [1] ))))))
        , (ModelElement 10 (PersonCh (Person "Zoe␣Zheng" 38 "Chief" "Female" (Just (TeammemberCh (Teammember "Chief" [5] [2] ))))))
        , (ModelElement 11 (PersonCh (Person "May␣Falkner" 27 "Staff" "Female" (Just (TeammemberCh (Teammember "Staff" [5] [2] ))))))
        , (ModelElement 12 (PersonCh (Person "Lindsey␣Scott" 36 "Chief" "Female" (Just (TeammemberCh (Teammember "Chief" [6] [3] ))))))
        , (ModelElement 13 (PersonCh (Person "James␣Pattern" 32 "Salesman" "Male" (Just (TeammemberCh (Teammember "Salesman" [6] [3] ))))))
        ]
```

Figure 7: Example of a Model in Haskell

V. Tools Support within Eclipse

In this section we describe tool support from a user's perspective through the case study presented in Section II. Haskell OCL [11] takes as an input a metamodel, together with their corresponding OCL invariants, and a model in which the invariants must be verified in order to check the conformance relation. These models are expressed using Eclipse OCLa con.

The difference between both tools is how the evaluation of invariants is performed: Eclipse OCL uses its own Java-based OCL interpreter, and we perform a functional interpretation following the process depicted in Figure 4. Our Haskell OCL plugin integrates both options in a menu, as shown in Figure 8, allowing to obtain the result in the most convenient way.

The results view in console (when executing the functional interpretation) is not user-friendly, as shown in Figure 8. It is just a list of ordered results, one for each invariant, such that in each case the result value of the interpretation is shown (for invariants, one of the four values of the semantics of OCL is expected). It can be noticed that, in this case, the second invariant results in the value `Val False`, i.e.: it is not satisfied. Besides it is possible to generate more detailed results with descriptive messages, we benefit from the use of the Eclipse OCL reporting tool (Validity View), as shown in Figure 10. Since both, Eclipse OCL and Haskell OCL, use the same inputs and provide the results in the same way, all the evaluation process (in the lower levels of Figure 4) can be completely transparent for the user.

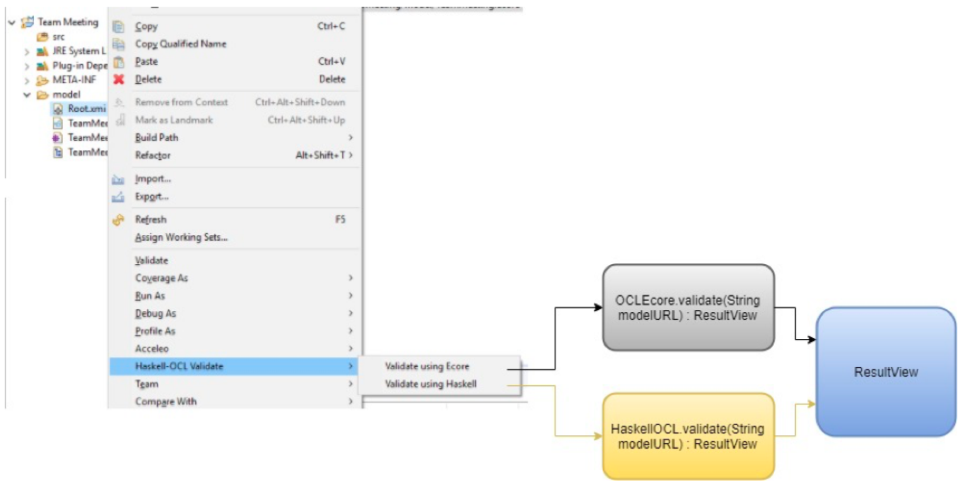
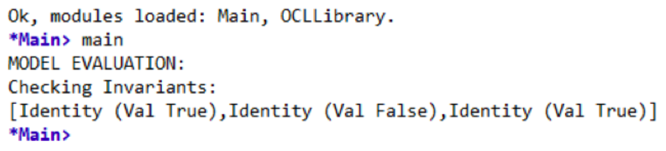


Figure 8: Haskell OCL validation



```
Ok, modules loaded: Main, OCLLibrary.
*Main> main
MODEL EVALUATION:
Checking Invariants:
[Identity (Val True),Identity (Val False),Identity (Val True)]
*Main>
```

Figure 9: OCL evaluation results

As a concrete example, if we evaluate the model of the case study, in Figure 3, with respect to the invariants defined in Figure 2, we got that every invariant is satisfied. The results are those shown in the Validity View depicted in Figure 10a. For each result a colored icon is used. In this case, a green tock is used in order to express a successful execution.

By changing the model, we can make the evaluation of the invariant `Inv3` fail. As shown in Figure 11, we can add a new team member named `John Smith`, which does not belongs to the `Sales` team, to the team meeting `Daily Meeting - Sales Dept`. In this way, when evaluating this new model, the third invariant fail, thus the Validity View reports that with a yellow attention icon, whilst the others are still satisfied, as shown in Figure 10b.

Since the generated Haskell file is available for experienced functional users, it is possible to directly change the representation of the model and evaluate the invariants using the Haskell perspective, without performing the whole transformation process. This can be useful when working with big models. However, it could be useful to have reverse engineering capabilities in order to keep models synchronized.

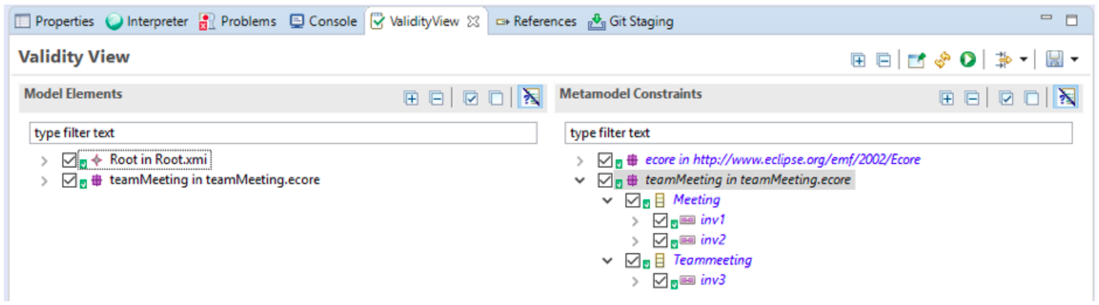
As an example, we can modify the model in Figure 12 such that the team meeting `Daily Meeting - Sales Dept` is expected to start (time 9) after it ends (time 8). By running again the evaluation of the constraints, we get that the invariant `Inv1` is not satisfied.

VI. Related Work and Discussions

The use of other semantic domains for the interpretation of OCL is usual for providing semantics and a formal environment for verification. In this context, many proposals provide a shallow embedding of the OCL into, e.g.: rewriting logic [15], constructive type theory [16], higher-order logic[17] and theory of institutions [18]. These works neither consider functional aspects nor a functional interpretation of such aspects. The more related works are our own. In [19] we propose the representation of MDE elements using Attribute Grammars which are expressed as Haskell expressions. We addressed the inclusion of OCL expressions for structural and semantic conformance checking, but we did not exhaustively study its representation. In [10] we explored the use of Haskell as an interpreter for OCL and provided insights on how advanced aspects could be supported, providing the basis for this current paper.

There are many OCL editors and interpreters, as those listed in the OCL Portal[4]. The most popular tools are Eclipse OCL [12] and Dresden OCL [20]). These two tools provide a (almost) complete support of the OCL language, not only for expressing invariants, but also for other used as pre- and post-conditions on operations. They are also focused on providing a direct representation of model-oriented features, which are useful in a wider model-driven environment. An interesting case is the one of HOL-OCL [17], an interactive proof environment for the OCL based on a formalization using Isabelle/HOL (a higher-order logic instance of the interactive theorem prover Isabelle) of a core part of OCL. This work is focused on a formal treatment of the key elements of the language rather than a complete implementation of it. It could be desirable indeed to examine the relation between our definitions and this work since Isabelle/HOL can be considered a functional programming language. A broader empirical assessment of our tool with respect to these other proposals is devised as future work.

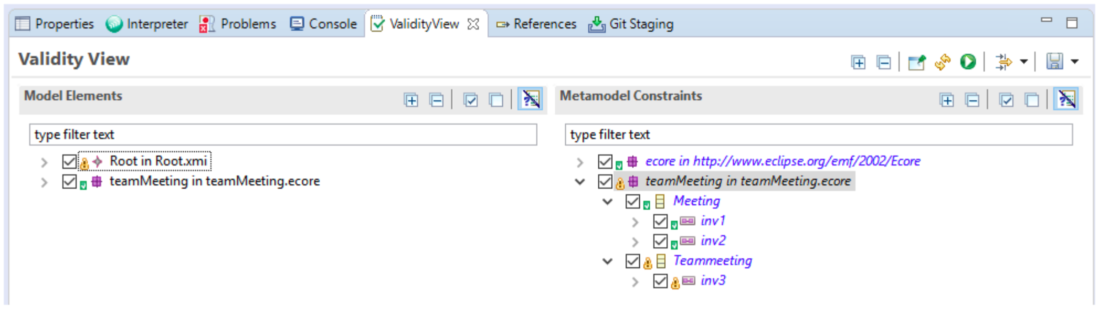
[4]OCL Portal: https://www-st.inf.tu-dresden.de/ocl/



Validity View

Model Elements	Metamodel Constraints
type filter text	type filter text
Root in Root.xmi	ecore in http://www.eclipse.org/emf/2002/Ecore
teamMeeting in teamMeeting.ecore	teamMeeting in teamMeeting.ecore
	Meeting
	inv1
	inv2
	Teammeeting
	inv3

(a) All invariants are satisfied

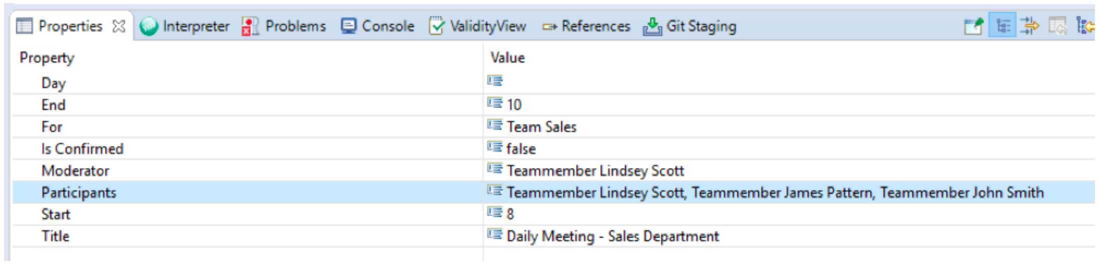


Validity View

Model Elements	Metamodel Constraints
type filter text	type filter text
Root in Root.xmi	ecore in http://www.eclipse.org/emf/2002/Ecore
teamMeeting in teamMeeting.ecore	teamMeeting in teamMeeting.ecore
	Meeting
	inv1
	inv2
	Teammeeting
	inv3

(b) Invariant (`Inv3`) is not satisfied

Figure 10: Eclipse OCL Validity View



Property	Value
Day	
End	10
For	Team Sales
Is Confirmed	false
Moderator	Teammember Lindsey Scott
Participants	Teammember Lindsey Scott, Teammember James Pattern, Teammember John Smith
Start	8
Title	Daily Meeting - Sales Department

Figure 11: Change on the participants of the meeting

```
example = Model
        [ (ModelElement 0 (RootCh (Root [7,8,9,10,11,12,13] [1, 2, 3] [4, 5, 6] )))
        ...
        , (ModelElement 3 (MeetingCh (Meeting "Daily␣Meeting␣–␣Sales␣Dept" "19/05/2018" 9 8 False [12,13] 12 []
                                             (Just (TeammeetingCh (Teammeeting 6))))))
        ...
        ]
```

Figure 12: Team Meeting model in Figure 7 (start and end time are swapped)

A. Compliance with the OCL and Limitations

It is useful to know the current level of compliance with the OCL standard and the limitations in order to state that its interpretation is comparable to the ones provided in more mature proposals, and also that it represents an extension of the original work in [10]. In general terms, it can be seen in Table I that our Haskell OCL proposal currently provides an almost complete representation of OCL as a query language and to specify invariants on classes and types in metamodels. The OCL elements not supported are those less used or related to metamodeling issues. With respect to the work in [10], besides we made some adjustments, we keep the same Haskell representation for models, metamodels and invariants, and we basically improved the OCL library.

We can state that most of the limitations correspond with the fact that we are supporting what is called Essential OCL, which provides the core capabilities for expressing invariants on models. In fact, Eclipse OCL supports parsing of other OCL aspects as pre- and post- conditions, messages and states, but does not provides any interpretation for them, since it is also focused on Essential OCL. Our biggest limitation, is not the support of OCL itself, but the support of other model constructs, in particular: composition of associations, ordered associations, operations on types, multiple inheritance, association classes and qualified associations. However, we postpone their interpretation since they are not focused (or not commonly used) on metamodels, but on other type of models (e.g.: class diagrams). Nevertheless, we need to continue developing the interpreter, and there are some OCL aspects that could be not easily represented (e.g.: tuples without a fixed length and heterogeneous collections) so they deserve further analysis. Moreover, it will be desirable to essay an extension of the interpreter to capture more expressive error messages, related to the work in [21].

VII. Conclusions and Future Work

In this paper, we presented how a functional perspective on the interpretation of OCL can be supported in practice, with respect to its use for expressing invariant conditions in models within the MDE paradigm. This approach allows to evaluate in a unified way several proposals that already exist in the scientific community, as well as to collaborate with the migration of functional aspects to the MDE paradigm.

We developed a predefined functional version of the OCL library, improving the one defined in [10], and we provided an Acceleo-based automatic transformation of models, metamodels and OCL invariants into Haskell for the interpretation of OCL. In this way, we showed that the functional infrastructure can be predefined and automatically generated. Besides there are some aspects still outside scope, and that some others could be not easily interpreted, the infrastructure covers a huge portion of OCL. In this context, we are working on the evaluation of advanced proposals, e.g.: extending the OCL with pattern matching and lambda expressions. Moreover, we need to consider other OCL uses, e.g.: for expressing constraints, pre- and post-conditions on operations, and for supporting the definition of model transformations.

We also presented Haskell OCL, an Eclipse-based tool support for final users, allowing to both: generating the infrastructure and also checking invariants, in such a way that the internal complexity is transparent to them. The use of Eclipse OCL as a base project allows us to avoid the complexity of parsing and type checking issues. Besides our proposal tends to be a sandbox for experimentation and not for professional use, it can be used in practice for real projects. A benchmark comparison between our interpreter and others is also of interest in this sense. Some important concerns that might be addressed are how the tool behaves on some (larger) real world examples, e.g.: dealing with performance concerns, and how the lack of features with respect to related approaches might or might not affect the user's ability to model these examples using the tool. Haskell OCL can also be improved in several ways: by capturing more expressive error messages from the functional evaluation of constraints; by adapting the tool to use OCL as a query language (which does not require any changes in the infrastruture), or other OCL uses (which indeed require further work for extending the functional infrastructure); and by applying a reverse engineering process in order to keep models synchronized when changes are made within the Haskell representation.

References

[1] S. Kent, "Model-driven engineering," in *IFM*, ser. Lecture Notes in Computer Science, vol. 2335. Springer, 2002, pp. 286–298.

[2] OMG, "Object Constraint Language," Object Management Group, Spec. V2.4, 2014.

[3] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[4] OMG, "Model Driven Architecture," Object Management Group, Spec. V2.0, 2014.

[5] T. Clark, "OCL pattern matching," in *Proc. OCL Workshop*, ser. CEUR Workshop Proceedings, vol. 1092. CEUR-WS.org, 2013, pp. 33–42. [Online]. Available: http://ceur-ws.org/Vol-1092/clark.pdf

[6] E. Willink, "Ocl 2.5 plans," presentation in the 14th Intl. Workshop on OCL and Textual Modelling, 2014.

[7] M. Tisi, R. Douence, and D. Wagelaar, "Lazy evaluation for OCL," in *Proc. 15th Intl. Workshop on OCL and Textual Modeling*, ser. CEUR Workshop Proceedings, vol. 1512. CEUR-WS.org, 2015, pp. 46–61.

[8] S. P. Jones, Ed., *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, September 2002.

[9] I. Kurtev, J. Bezivin, and M. Aksit, "Technological spaces: An initial appraisal," in *International Symposium on Distributed Objects and Applications*, 2002.

[10] D. Calegari and M. Viera, "On the functional interpretation of OCL," in *Proc. of the 16th Intl. Workshop on OCL and Textual Modelling co-located with 19th Intl. Conf. on Model Driven Engineering Languages and Systems (MODELS 2016).*, ser. CEUR Workshop Proceedings, vol. 1756. CEUR-WS.org, 2016, pp. 33–48. [Online]. Available: http://ceur-ws.org/Vol-1756/paper03.pdf

[11] G. Sintas, L. V. Lutz, D. Calegari, and M. Viera, "HaskellOCL: A Haskell-based functional interpreter for the Object Constraint Language," 2018. [Online]. Available: https://gitlab.fing.edu.uy/open-coal/haskellOCL

[12] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*. Boston, Massachusetts: Addison-Wesley Professional, 2008.

[13] B. Demuth, "Ocl (Object Constraint Language) by example. lecture at mine summer school, nida," 2009. [Online]. Available: https://st.inf.tu-dresden.de/files/general/OCLByExampleLecture.pdf

[14] G. Huet, "The zipper," *J. Funct. Program.*, vol. 7, no. 5, pp. 549–554, Sep. 1997. [Online]. Available: http://dx.doi.org/10.1017/S0956796897002864

[15] A. Boronat and J. Meseguer, "Algebraic semantics of OCL-constrained metamodel specifications," in *TOOLS (47)*, ser. LNBIP, vol. 33. Springer, 2009, pp. 96–115.

[16] D. Calegari, C. Luna, N. Szasz, and A. Tasistro, "A type-theoretic framework for certified model transformations," in *13th Brazilian Symposium Formal Methods*, ser. LNCS, vol. 6527. Springer, 2010, pp. 112–127.

[17] A. D. Brucker, F. Tuong, and B. Wolff, "Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5," *Archive of Formal Proofs*, vol. 2014, 2014. [Online]. Available: http://afp.sourceforge.net/entries/Featherweight_OCL.shtml

[18] D. Calegari, T. Mossakowski, and N. Szasz, "Heterogeneous verification in the context of model driven engineering," *Sci. Comput. Program.*, vol. 126, pp. 3–30, 2016.

[19] D. Calegari and M. Viera, "Model-driven engineering based on attribute grammars," in *Proc. 19th Brazilian Symposium Programming Languages*, ser. LNCS, vol. 9325. Springer, 2015, pp. 112–127. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-24012-1_9

[20] C. Wilke, M. Thiele, and B. Freitag, "Dresden OCL - manual for installation use and development," TU Dresden, Tech. Rep., 2009-2011.

[21] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "On the evolution of OCL for capturing structural constraints in modelling languages," in *Rigorous Methods for Software Construction and Analysis*, ser. LNCS, vol. 5115. Springer, 2009, pp. 204–218. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11447-2_13