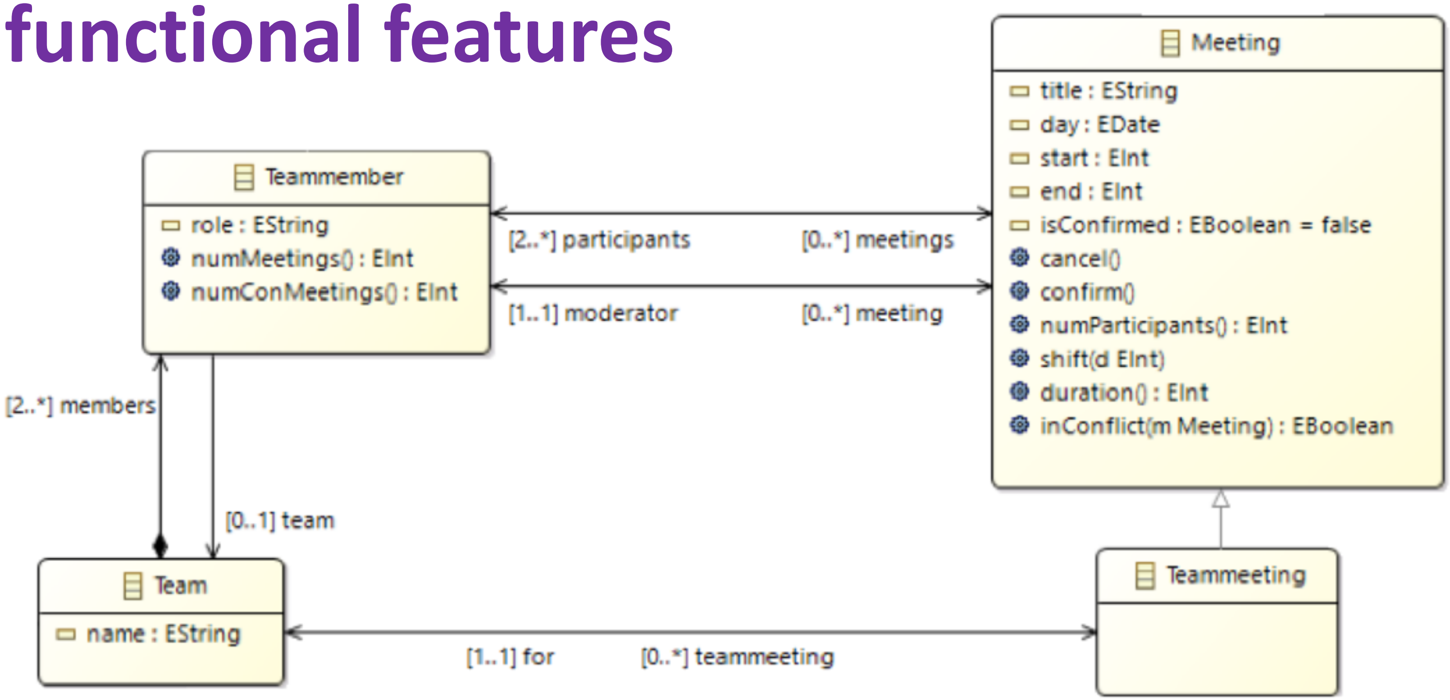
19th International Workshop in OCL and Textual Modeling

Experimenting with functional features of the Object Constraint Language

Daniel Calegari, Marcos Viera
Universidad de la República, Uruguay
{dcalegar,mviera}@fing.edu.uy

context

OCL combines model-oriented and functional features




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

context

we have explored a functional approach to support the construction of an OCL interpreter (for invariants)

- **a sandbox for experimentation**
- **an EDSL in Haskell + a tool Haskell OCL**
- **the functional infrastructure can be predefined and automatically generated**

D. Calegari and M. Viera, "On the functional interpretation of OCL," in *Proc. of the 16th Workshop on OCL and Textual Modelling*, ser. CEUR vol. 1756, pp. 33–48, 2016

G. Sintas, L. Vaz, D. Calegari, M. Viera. "Model-Driven Development of an Interpreter for the Object Constraint Language". Proc. CLEI 2018, IEEE: 120-128

objectives

experiment with functional features proposed by the scientific community (e.g., pattern matching, lazy evaluation, lambda abstractions, etc.)

how OCL interpretation benefits from such an encoding and its limitations

- **not focusing on proposing new features**
- **not on making any comparison w.r.t. technological support for the language**

interpretation of models & meta

metamodel and models are automatically translated to Haskell datatypes/values

```
data ModelElement      = ModelElement Int ModelElementChild
data ModelElementChild = PersonCh  Person | ...
data Person            = Person String Int String String (Maybe PersonCh)
data PersonCh          = TeammemberCh Teammember
data Teammember        = Teammember String [Int] [Int] [Int]
...
name :: Cast Model Person_ a => Val a -> OCL Model (Val String)
name a = upCast _Person a >>= pureOCL (\(Person x _ _ _ _) -> return (Val x))
```

Boilerplate code is defined to navigate through a model easily/uniformly

interpretation of invariants

invariants are automatically translated to Haskell functions mimicking its structure

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

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

monad (OCL m a): computations within a model *m*, returning a value *a* (OCL four-valued logic)

the OCL library

it is predefined and provides an almost complete support for invariants and queries

	Haskell OCL	Eclipse OCL
OCL Constructs & Expressions		
`context` / `inv` (invariant condition and its context)	✓	✓
`init` (initial value of an attribute or association role)	✓	✓
`derive` (derived attribute or association role)	✗	✓
`def` (new attribute or query operation)	✓	✓
`package` (package to which OCL expressions belong)	~ [a]	✓
`self` (contextual instance)	✓	✓
`if-then-else` / `let-in` expressions	✓	✓
Navigation (through attributes, association ends, etc.)	~ [b]	✓
OCL Standard Library (types and operations)		
`Boolean/Integer/Real/String` types	✓	✓
`UnlimitedNatural` type	✗	✓
`OCLAny` type (supertype of all OCL types)	~ [c]	~
`OCLVoid` type (one single instance `undefined`)	~ [d]	✓
`Tuple` type	✓	✓
`Collection` types (`Set`, `OrderedSet`, `Bag`, and `Sequence`)	✓	✓
`Collection` operations	~ [e]	✓

Disclaimer

the following requires strong functional programming understanding

lets discuss some interesting findings and lessons learned from the experience

- functional support
- model-oriented vs functional
- monadic interpretation

functional support

functions are first-class citizens in Haskell

OCL defines something like functions (e.g., when defining a let expression)

collection operators use lambda abstractions e.g.,

```
reject p = select (notOCL . p)
```

functional support

what about higher-order functions?

```
-- the moderator is the one with highest priority
context Meeting inv:
let priority(Set(Teammember)) : Teammember = ... ,
    getModerator(m:Meeting,
       p:(Set(Teammember) -> Teammember))
       : Teammember = p(m.participants)
in self.moderator = getModerator(self, priority)
```

there are some problems, e.g., `flatten` is not easily supported when there are multi-type elements within the collection

functional support

is it feasible / valuable to have an extensive support for functions in OCL

model-oriented vs functional

hierarchical typing and identities introduce the main mismatch problems

some functional boilerplate can be generated to minimize the impact

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

¿what about a multi-paradigm interpreter?

model-oriented vs functional

Sigma is an OCL EDSL implemented in Scala

since Scala is both functional and object-oriented, their embedding does not have to deal with mismatch problems

however, Sigma OCL expressions are not effect-free, and formal reasoning is much more difficult than in a purely functional approach

model-oriented vs functional

is it feasible / valuable to have a balance between model-oriented and functional interpretation for OCL

monadic interpretation

a `Reader` monad "silently" passes a model through the sequences of computations

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

in some cases OCL could be benefited from the introduction of controlled side effects

monads can be composed for adding these behaviors in a modular way

monadic interpretation

`Error` monad represents computations which may fail or throw exceptions

```
type OCLError m a = ErrorT String (OCL m a)
invariant1 = context _Meeting [inv2]
inv2 self = do res <- ocl self |.| participants ...
               if res |==| oclVal True
               then return res
               else throwError ”There was an error”
```

`State` monad consumes a state and produce both a result and an updated state (e.g., a repaired model)

monadic interpretation

is it feasible / valuable to define a modular effects mechanism for OCL (e.g., inspired by monads and monad transformers)

conclusions & future work

we experimented with functional features proposed for OCL and provided a different perspective of OCL as a Haskell EDSL

the expressions and their interpretation are clean: modular, abstract and extensible

there are many challenges, e.g., laziness can improve performance but also adds memory overhead

19th International Workshop in OCL and Textual Modeling

Experimenting with functional features of the Object Constraint Language

Daniel Calegari, Marcos Viera
Universidad de la República, Uruguay
{dcalegar,mviera}@fing.edu.uy