Implementation of a CVL to Clafer transformation

Tom Wijsman

Model Driven Engineering 2014-2015
University of Antwerp

Abstract

Clafer and CVL are currently developing modeling languages that combine modeling languages with variability. They tackle historical problems when introducing variability in existing models. Clafer is textual and concise; CVL is visual and an expansion. In order to make comparisons, interactions and migrations between these two modeling languages more possible; a transformation is implemented that transforms CVL models to Clafer. For completeness, note that the transformation in the opposite direction already exists in the Clafer Compiler.

Keywords: Choices, Clafer, Class, Constraint, CVL, EGL, ETL, Feature, Feature Model, Generation, Language, Reference, Representation, Textual, Transformation, Unification, Variability, Variation, Visual

1. Introduction

An implementation of a CVL to Clafer transformation is outlined. This implementation is then verified by transforming back from Clafer to CVL.

1.1. Context

For the course Model Driven Engineering at the University of Antwerp a project is studied and implemented. This project incorporates most of the knowledge exchanged at the course lectures and practicums, which covers (but is not limited to) theory and practice about modeling languages, syntax, semantics and transformations.

Email address: `Tom.Wijsman@student.uantwerpen.be` (Tom Wijsman)

In the previous paper by Wijsman (2014) that introduced this project the CVL and Clafer languages were studied, their tools were explored and a summarizing look at the history of modeling languages with variability has been made. In that paper the problem has been described and future work has been planned. This paper implements that work as a solution to it.

1.2. Problem

Each language has its own representation by design; Clafer is textual, whereas CVL is graphical. Someone interested in such languages will compare representations and perhaps prefer one representation over the other. In interactions between people there can be interactions where one person uses Clafer, whereas the other person uses CVL or vice versa. It is also possible that someone needs to switch between Clafer and CVL. Being able to transform one representation into the other and vice versa could help for these comparisons, interactions and migrations.

Searching for existing solutions yields that the Clafer compiler can produce a CVL representation from a Clafer model; however, the transformation in the other direction from CVL to Clafer does not seem to exist. The transformation from CVL to Clafer outlined in this paper implements this.

1.3. Overview

First the transformation steps are enumerated in section 2, which are explained in detail in subsections 2.1 - 2.6. Then an example CVL to Clafer transformation result is given in section 3. This example will be verified by transforming the Clafer example result back to CVL and comparing them in section 4. A conclusion is given in section 5. Future work is discussed in section 6. The requirements, files and execution of the project's work are summarized in section 7.

2. CVL to Clafer transformation

Transformation from CVL to Clafer is done in multiple steps, which includes reuse of the generation language EGL and the transformation language ETL. The framework metaDepth for multi-level meta-modeling is used in intermediate steps. An overview of the steps that are performed:

1. identify a reasonable set of common features that exist in both CVL, Clafer and their constraint languages;

2. create an abstract and concrete visual syntax of CVL in AToMPM;
3. export a metaDepth model of CVL from AToMPM by using a AToMPM to metaDepth exporter;
4. transform the metaDepth model of CVL to Clafer by using ETL (Epsilon Transformation Language);
5. generate concise syntax in Clafer from the metaDepth model of Clafer by using EGL (Epsilon Generation Language).

2.1. Identification of common features

The following table lists the features of CVL and their Clafer equivalents:

CVL	**Clafer**
Variability Specification	Class/Feature Type
Variability Choice	Feature
Variability Variable	Type Definition
Optional VSpec	"?" behind the Class/Feature Type
Type of Variable	": Type" behind the Class/Feature Type
Group Cardinality	"xor" (1..1), "or" (1..*) or "x..y" (x..y) before the Class/Feature Type
Children of a VSpec	Indented one level deeper than the parent

For constraints a transformation between two small custom constraint languages has been invented, as a transformation between the whole constraint languages could be considered a project on its own.

2.2. Abstract syntax of CVL in AToMPM

In figure 1 the abstract syntax metamodel of CVL in AToMPM can be seen.

A Root class is added to give a notion of a starting point for the transformation. The whole metamodel revolves around the central VSpec (short for Variability Specification) class, which can either be a Choice (with a name) or a Variable (with a name and type) class. A VSpec can have VSpecs as children; either through a mandatory link, an optional link or through a

GroupCardinality class (with a minimum and maximum amount of connected children).

A Constraint class is added such that constraints can be connected to a VSpec. A connected Constraint allows to put constraints on the selection and assignment of the children of the VSpec. One constraint is given by the ChoiceConstraint class; which uses ”And” and ”AndNot” links to determine respectively which children can and can’t be chosen together. The other constraint is given by the ComparisonConstraint class; which allows to make comparisons between a child that is connected through the LHS link and a child that is connected through the RHS link.

Not shown in the figure is that the classes and links in the metamodel are also given various cardinalities, such that only sane CVL models can be made.

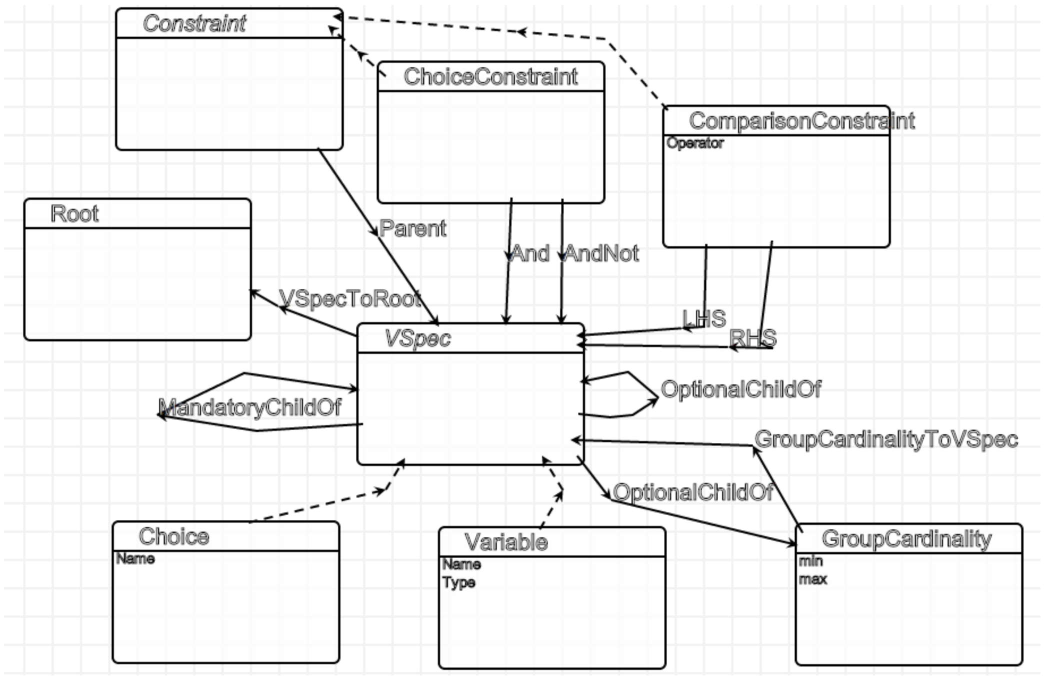

Figure 1: The abstract syntax metamodel of CVL in AToMPM.

2.3. Concrete visual syntax of CVL in AToMPM

For each class and link in the abstract syntax metamodel, a concrete visual syntax has been made to be able to visually create and represent a

CVL model.

In figure 2 you can see the concrete visual syntax for a root element in AToMPM. It is small, as in a CVL model there isn't supposed to be a visual root element visible; it is solely here to assist the transformation.

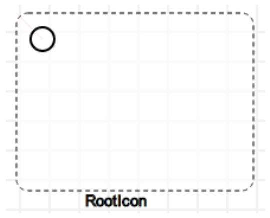

Figure 2: Concrete visual syntax for the root of CVL in AToMPM.

In figure 3 you can see the concrete visual syntax for the two variability specifications of CVL in AToMPM. A choice is a rectangle with round corners, which contains its name. A variable is an oval, which contains both its name and type separated by a colon. It is visually similar to the CVL representation.

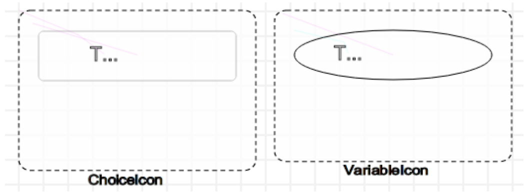

Figure 3: Concrete visual syntax for the variability specifications of CVL in AToMPM.

In figure 4 you can see the concrete visual syntax for the children links between variability specifications of CVL in AToMPM. A mandatory link is in black bold, whereas an optional link is in a thin gray; this makes the difference clear. It is however visually different from the CVL representation, as the dash-dotted line of CVL could not be reproduced in AToMPM.

In figure 5 you can see the concrete visual syntax for the group cardinalities of CVL in AToMPM. It shows the minimal and maximal cardinality. It is visually similar to the CVL representation.

In figure 6 you can see the concrete visual syntax for the custom constraints in AToMPM. A smaller rectangle has been chosen here to look significantly different from variability specifications. The type of constraint is

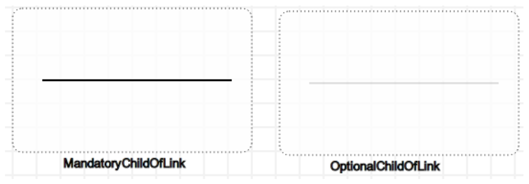

Figure 4: Concrete visual syntax for the variability specifications children links of CVL in AToMPM.

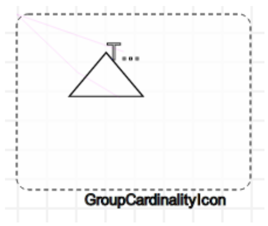

Figure 5: Concrete visual syntax for the group cardinalities of CVL in AToMPM.

shown inside the rectangle. For the comparison contraint additionally the comparison operator is shown inside the rectangle.

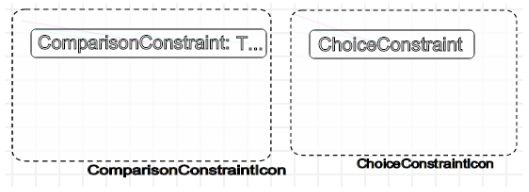

Figure 6: Concrete visual syntax for custom constraints in AToMPM.

In figure 7 you can see the concrete visual syntax for the custom choice constraint links in AToMPM. A green "And" link is used to denote inclusion. A red "AndNot" link is used to denote exclusion. They are therefore significantly different as to not be confused with other links.

In figure 8 you can see the concrete visual syntax for the custom comparison constraint links in AToMPM. Both links list the side of the comparison they link to; thus one link is connected to the LHS VSpec, whereas the other link is connected to the RHS VSpec.

2.4. Export of AToMPM CVL model to metaDepth

When exporting the metamodel in AToMPM and any models in AToMPM to metaDepth files, you get a metaDepth model of the CVL metamodel as

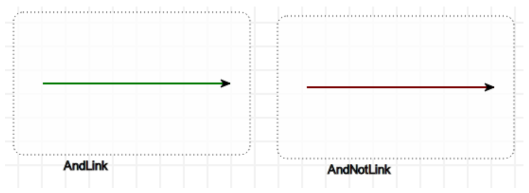

Figure 7: Concrete visual syntax for the custom choice constraint links in AToMPM.

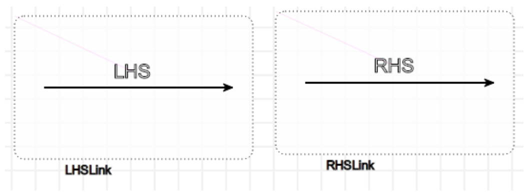

Figure 8: Concrete visual syntax for the custom comparison constraint links in AToMPM.

well as metaDepth models of the CVL models. The metaDepth model of the CVL metamodel serves as the metamodel for the metaDepth models of the CVL models.

2.5. Transformation to Clafer in metaDepth using ETL

```
Model ClaferMM {
        Node Link {
                src : Node;
                dst : Node;
        }

        Node Root {}

        Node Clafer {
                Name:String = "";
                Optional:boolean = false;
                Type:String = "";
                CardType:String = "";
                IsRootElement:boolean = false;
        }

        Node Constraint {
```

```
                        Constraint:String = "";
            }

            Node ClaferToRoot: Link {}
            Node ClaferToParent: Link {}
            Node ConstraintToClafer: Link {}
}
```

The Clafer metamodel in metaDepth has been kept simple and is shown in the code above. A Root node is again introduced to assist the transformation process. A Clafer node represents multiple CVL features, which are captured in its parameters (name, optional, type and cardinalities). The Constraint node allows constraints to be added. Besides those nodes, there is the ability to instantiate links between the nodes (between clafer and root, between clafer and a clafer and between clafer and a constraint).

The Clafer node also has an IsRootElement to be able to enumerate them. This is because a bug during the generation process requires referenced instantiated nodes to go before the instantiated nodes that reference them. As the instantiated Root element is put last, it is not seen during the generation process.

In the code listings that follow the ETL based transformation is explained in detail.

```
rule CVL2Clafer
  transform s : Source!Root
  to t : Target!Root {

  }

rule CVL2Clafer_C2R
  transform s : Source!VSpecToRoot
  to t : Target!ClaferToRoot {
    var c : new Target!Clafer;
    c.Name = s.src.Name;
    c.Optional = false;
    c.IsRootElement = true;

    s.src.enumChildren(c);

    t.src = c;
    t.dst = s.dst.equivalent();
  }
```

In the code listing above the CVL root and CVL root VSpecs are respectively transformed into a Clafer root and Clafer root clafers. The clafers' parameters are filled in and the clafers are linked to the Clafer root. Then the children of the VSpecs are enumerated recursively, as the visual tree design allows us to easily approach the child nodes in a recursive way.

```
operation Source!VSpec enumChildren(newself : Target!
    Clafer) {
  self.enumMandatoryChildren(newself);
  self.enumOptionalChildren(newself);
  self.enumGroupCardinalities(newself);
  self.enumConstraints(newself);
}
```

In the code listing above an enumeration happens for the mandatory and optional children of a VSpec, as well as the connected group cardinalities and constraints.

```
operation Source!VSpec enumMandatoryChildren(newself :
    Target!Clafer) {
  for (e in MandatoryChildOf.all.select(i | i.dst.name
      = self.name)) {
    var c : new Target!Clafer;
    c.Name = e.src.Name;

    if (e.src.isTypeOf(Source!Variable)) {
      c.Type = e.src.Type;
    }

    var n : new Target!ClaferToParent;
    n.src = c;
    n.dst = newself;

    e.src.enumChildren(c);
  }
}
```

In the code listing above, the mandatory children of a VSpec are enumerated. The parameters are set according to the corresponding VSpec (if it is a variable, take over its type) and a corresponding link is made.

The code listing for optional children is left out of this report, since it is similar and additionally sets the optional parameter to be true.

```
operation Source!VSpec enumGroupCardinalities(newself :
    Target!Clafer) {
  for (e in GroupCardinalityToVSpec.all.select(i | i.
     dst.name = self.name)) {
    if (e.src.min = "1" and e.src.max = "1") {
      newself.CardType = "xor";
    }
    else if (e.src.min = "1" and e.src.max = "*") {
      newself.CardType = "or";
    }
    else if (e.src.min = "0" and e.src.max = "*") {
      newself.CardType = "?";
    }
    else {
      newself.CardType = e.src.min + ".." + e.src.max;
    }

    for (f in OptionalChildOf.all.select(i | i.dst.name
        = e.src.name)) {
      var c : new Target!Clafer;
      c.Name = f.src.Name;
      if (newself.CardType = "?") {
        c.Optional = true;
      }

      if (f.src.isTypeOf(Source!Variable)) {
        c.Type = f.src.Type;
      }

      var n : new Target!ClaferToParent;
      n.src = c;
      n.dst = newself;
```

```

        f.src.enumChildren(c);
      }
    }
}
```

In the code listing above, the group cardinalities that are children of a VSpec are enumerated. The minimum and maximum cardinalities are used to create the right Clafer parameter; 1..1 corresponds to "xor", 1..* corresponds to "or", 0..* corresponds to that the clafer is optional and other cardinalities are kept as is. After this has been determined and set in the group cardinality parent's clafer, child clafer are made in a similar way as the parent of the group cardinality.

```
operation Source!VSpec enumConstraints(newself : Target
    !Clafer) {
  for (e in Parent.all.select(i | i.dst.name = self.
      name)) {
    if (e.src.isTypeOf(Source!ChoiceConstraint)) {
      var c : new Target!Constraint;
      c.Constraint = "";

      var first : Boolean = true;

      for (a in And.all.select(i | i.src.name = e.src.
          name)) {
        if (first) {
          first = false;
          c.Constraint = a.dst.Name;
        }
        else {
          c.Constraint = c.Constraint + " && " + a.dst.
              Name;
        }
      }

      for (a in AndNot.all.select(i | i.src.name = e.
          src.name)) {
        if (first) {
          first = false;
          c.Constraint = a.dst.Name;
        }
        else {
          c.Constraint = c.Constraint + " && !" + a.dst
             .Name;
        }
      }

      var n : new Target!ConstraintToClafer;
      n.src = c;
      n.dst = newself;
    }
    else if (e.src.isTypeOf(Source!ComparisonConstraint
       )) {
      var c : new Target!Constraint;
      c.Constraint = "";

      for (a in LHS.all.select(i | i.src.name = e.src.
         name)) {
        c.Constraint = a.dst.Name;
      }

      c.Constraint = c.Constraint + " " + e.src.
         Operator + " ";

      for (a in RHS.all.select(i | i.src.name = e.src.
         name)) {
        c.Constraint = c.Constraint + a.dst.Name;
      }

      var n : new Target!ConstraintToClafer;
      n.src = c;
      n.dst = newself;
    }
  }
}
```

In the code listing above, the constraints connected to a clafer are enumerated. For each constraint, the type is determined after which the other connections to the constraint are enumerated and uses to build a constraint string. This constraint string is set in the constraint node, such that it can be inserted right away when the generation of the constraints is done. A link between the clafer and the constraint is made.

2.6. Generation in metaDepth using EGL to Clafer

Now that there is a Clafer model in metaDepth, the next step is to generate the Clafer text from the Clafer model in metaDepth.

In the code listings that follow the EGL based generation is explained in detail.

```
[% for (c in Clafer.all.select(x | x.IsRootElement = true)) { %]
[%=c.printClafer() %]
        [%=c.enumChildren() %][% } %]
```

In the code listing above all the root clafers are enumerated, printed out and their children are enumerated.

```
[%
@template
operation Clafer enumChildren() {
        for (cp in ClaferToParent.all.select(x | x.dst = self)) {
                %][%=cp.src.printClafer() %]
        [%=cp.src.enumChildren() %]
                [%
        }
        for (cc in ConstraintToClafer.all.select(x | x.dst = self)) {
                %][ [%=cc.src.Constraint %] ][%="\n" %][%
        }
}
```

In the code listing above both the child clafers and the child constraints of a parent clafer are enumerated. The child clafers are enumerated recursively.

```
@template
operation Clafer printClafer() {
        %][%=self.printCardType() %][%=self.Name %][%=
            self.printType() %][%=self.printOptional()
            %][%
}
```

In the code listing above the clafer's cardinality, name, type definition and whether it is optional is printed out.

```
@template
operation Clafer printCardType() {
        if (self.CardType <> "?" and self.CardType <>
            "") {
                var literalPostfix : String = " ";
                %][%=self.CardType %][%=literalPostfix
                    %][%
        }
}

@template
operation Clafer printOptional() {
        if (self.Optional = true) {
                %]?[%
        }
}

@template
operation Clafer printType() {
        if (self.Type <> "") {
                var literalPrefix : String = " : ";
                %][%=literalPrefix %][%=self.Type%][%
        }
}
%]
```

In the code listing above the textual syntax to be printed for the cardinality, optional "?" and type definiton are implemented.

3. Example

In figure 9 a CVL model in AToMPM that features all the implemented CVL and Clafer features can be seen.

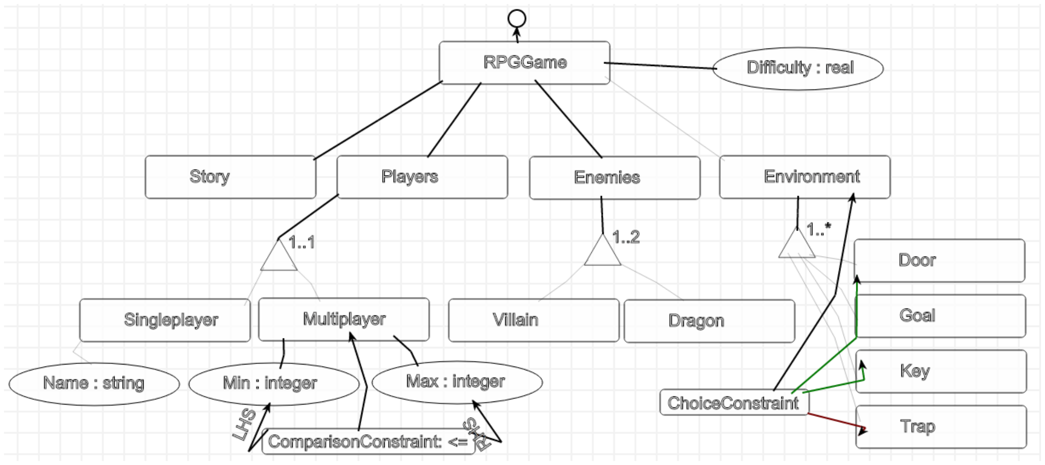

Figure 9: An example RPGGame of CVL in AToMPM.

When using the transformation steps outlined in this paper on this example CVL model in AToMPM, the following expected Clafer textual representation is obtained:

```
RPGGame
        Story

        xor Players
                Singleplayer
                        Name? : String

                Multiplayer
                        Min : Integer

                        Max : Integer

                        [ Min <= Max ]

        1..2 Enemies
                Villain

                Dragon

        Difficulty : Float

        or Environment?
                Door

                Goal

                Key

                Trap

                [ Key && Door && !Trap ]
```

4. Verification

As the Clafer Compiler uses a transformation in the other direction to visualize a Clafer model by producing a CVL model in a DOT file; the transformation can be verified by transforming the concise syntax of Clafer that the transformation produced back to CVL, then do a visual comparison of both. They should have the same structure to pass verification.

The commands used to turn a Clafer.cfr file into a CVL.png figure are:

```
clafer -m=cvlGraph Clafer.cfr
dot -Tpng -o CVL.png Clafer.cvl.dot
```

In figure 10 a CVL model that is tranformed from the example Clafer textual representation result can be seen. Apart from a bug with the constraint mapping onto a single node, it looks similar enough. In other words, the transformation example passes the verification.

5. Conclusion

A common subset of features between CVL and Clafer exists that is large enough for a successful transformation of a large set of CVL models to Clafer.

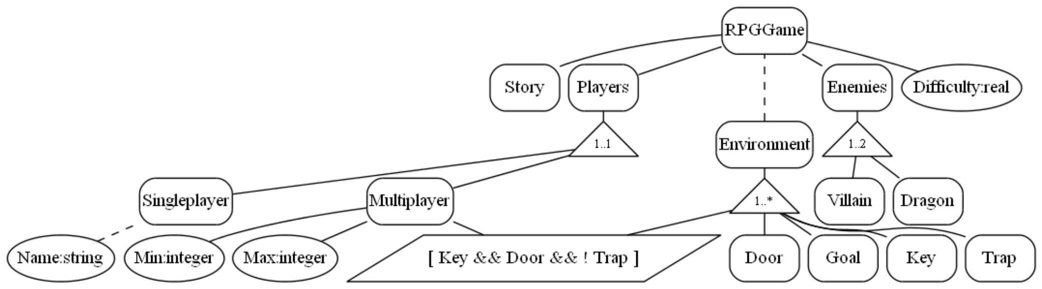

Figure 10: Verification by using the Clafer Compiler to transform the example RPGGame from Clafer back to CVL.

6. Future work

Future work could study the similarities and differences of the instantiation of CVL and Clafer, such that the variation points of CVL can be transformed to similar concepts (for example, an attempt to express it as constraint syntax) or methods (for example, an attempt at manipulating the instance generator) in Clafer.

CVL constraints (which are OCL based) and Clafer constraints (which are Alloy based) could be studied and compared in order to create a more complete transformation from CVL constraints to Clafer constraints.

Both these future work suggestions could be seen as projects on their own; because of this project's scope, they were limited or left out of this project.

7. Files

7.1. Requirements

For generation and transformation metaDepth 2.0 is required, metaDepth.jar needs to put in the metaDepth folder.

For the transformation in the reverse direction from Clafer to CVL, Clafer Tools 0.4.0 is required, as well as "dot" from Graphviz with PNG support.

7.2. Contents

7.2.1. AToMPM

- **Formalisms/CVL:** The abstract syntax and concrete visual syntax metamodel files.
- **Models/CVL:** Example models: CVL was used during development, RPGGame is the final example.

7.2.2. metaDepth

The following files are the intermediary steps and results of the transformation and generation:

- **CVLMM.mdepth:** The exported CVL metamodel in metaDepth.
- **CVL.mdepth:** The exported RPGGame CVL model which uses CVLMM in metaDepth.
- **ClaferMM.mdepth:** A Clafer metamodel made in metaDepth.
- **Clafer_empty.mdepth:** An empty Clafer model which uses ClaferMM in metaDepth.
- **CVL2Clafer_transformation.etl:** An ETL transformation from CVL to Clafer in metaDepth.
- **Clafer.mdepth:** The transformed Clafer model in metaDepth.
- **Clafer_generation.egl:** An EGL generation from Clafer in metaDepth to Clafer in its textual representation.
- **Clafer.cfr:** The resulting Clafer textual representation.

The following files support running the transformation and generation:

- **CVL2Clafer_cmdlist:** Runs the metaDepth instructions to do the ETL transformation.
- **Clafer_main.egl:** Main file that runs the EGL generation.
- **Clafer_cmdlist:** Runs the metaDepth instructions to do the EGL generation.
- **run.bat:** Runs the transformation and the generation; as an intermediary step, Clafer.mdepth is corrected to be further processed.

7.3. Execution

The transformation and generation can be done by running `run.bat`.

8. Bibliography

Wijsman, T., 2014. Introduction to a CVL to Clafer transformation project.