Metamodel-independent modularisation of models with MetaMod

Ana Maria Şutii
Eindhoven University of Technology, Netherlands

Tom Verhoeff
Eindhoven University of Technology, Netherlands

Mark van den Brand
Eindhoven University of Technology, Netherlands

ABSTRACT
In today’s landscape of more and more software-driven functionalities, spanning more and more fields, model-driven engineering (MDE) promises to ease the development of software. To accomplish this goal, MDE employs domain-specific languages (DSLs). The problem is that, on one hand, DSLs are not easy to create, and, on the other hand, as a result of the increased software-driven functionalities, they need to deal with bigger models. In dealing with these big models, modularity mechanisms are employed regularly by DSLs. These mechanisms need to be introduced over and over again into the developed DSLs, adding to the effort of creating them. To ease the development of DSLs, we propose to introduce a modularisation of models that is independent of the DSLs. We do so via two mechanisms, groups and fragment abstractions, that comprise many modularity use cases found in DSLs. These two mechanisms have been implemented in a prototype tool, MetaMod.

1. INTRODUCTION
With the increased use of software in all fields of business, the need for automating software construction increases. In the context of today’s software development challenges, model-driven engineering promises to bring improvements. MDE raises the level of abstraction in programming languages and brings them closer to the domain of operation. In MDE, domain-specific languages - computer programming languages, usually declarative, with limited expressiveness and focused on a particular domain - are key players. In this context, the structure of a DSL is captured in a metamodel and programs written using the DSL are models that conform to the metamodel of the DSL.

One drawback of DSLs is that they require a significant amount of work to be created. Moreover, the large models that we need to deal with nowadays complicate their creation even more. Languages need to cater for these big models by introducing modularity mechanisms into the languages themselves. Modularity mechanisms need to be added to the languages over and over again. This adds to the effort of designing and implementing the language considerably.

That is why, we introduced generic mechanisms that cover most modularity use cases and that can be used in programs of newly developed DSLs by default. They considerably ease the burden of the language developers because they do not need to add constructs for modularity anymore in the metamodels of the DSLs. Of course, we do not cover those modularity mechanisms that need semantical information from the language itself. Compared to related work, in our approach one does not need to modify the metamodel of the language [3] or to prescribe the modularisation in the metamodel itself [1].

2. METHOD
To this end, we built MetaMod. MetaMod is a research tool for modularity experiments in languages and models of the languages. The modularity mechanism provided in MetaMod is based on groups and parameterized model fragments [2]. The groups are meant to organize model elements and due to their semantics, they are self-contained. The parameterized model fragments (also called fragment abstractions) are a generalization of the groups and are based on model element substitution. One novel aspect in fragment abstractions is the fact that we leveraged the substitution mechanism in untyped lambda calculus by combining untyped lambda calculus with modeling elements. An example of a group can be seen in Figure 1 and an example of a fragment abstraction can be seen in Figure 2.

MetaMod allows that groups and fragment abstractions can be used in any newly developed DSL with no further modifications to the DSLs.

3. REFERENCES
Figure 1: Simplified model of the Rotterdam train. Two sections in the city, Slinge-Binnenhof and Binnenhof-Central, are captured in it. The sections reside in different sub-groups. Each section is made of two segments with an intermediate point in between the endpoints of the sections. The groups are shown using the visual syntax of MetaMod.

```
segBetweenEndPoints = forall EndPoint1.
forall EndPoint2.
forall SegLength.

type model TrainTracks

RotterdamTrain group {

  Slinge-Binnenhof group {
    Binnenhof : Point
    IPointSB : Point
    Slinge : Point

    Segments:
    IPointSB_Binnenhof : Segment
    Slinge_IPointSB : Segment

  }

  Binnenhof-Central group {
    Binnenhof : Point
    IPointBC : Point
    Central : Point

    Segments:
    IPointBC_Central : Segment
    Central_IPointBC : Segment

  }

}
```

Figure 2: Fragment abstraction that creates a segment with the endpoints received as parameters and the segment length received as a parameter. The fragment abstraction is shown using the visual syntax of MetaMod.

```
segBetweenEndPoints = forall EndPoint1.
forall EndPoint2.
forall SegLength.

type model TrainTracks

RotterdamTrain group {

  Segments:
  Seg : Segment

}
```