**Ideas regarding practical implementation**

Difference – for us deep modeling is just a tool to express entity-relationship hierarchies. We are not trying to embed the concept of deep instances into models. We do not want to extend the notion of deep modeling further. Our key approach and difference are that we support variability. That means that we have some snapshots of a big model at runtime and the instance is attached to this snapshot. In other words, in our case, there are two model spaces – model space (defining models) and user space (instances). When an instance has been created the model that is used at this time is forked and related to this instance. That means that if the same model is changed later at some, the instance will stay invariant because it contains enough knowledge to construct its slots, values, and links. There is a lot of inconsistency -> deep instance is not consistent with the current model but consistent with its model. This allows us to achieve variability. Again, we are inconsistent on user-model space but we are consistent in user space.

To make constraints consistent with the current approach it would make sense to define constraints in model space but as soon as an instance is created -> those constraints should be copied to user space. This means that constraints will be also consistent in user space only to its model defined inside of user space.

Constraints should make sense of course. If something is invalid, we should consider either soft or strict constraint evaluation. Soft constraint evaluation means that if some constraint is invalid, we still allow a user to execute an operation but we warn him. Strict constraint evaluation means that we do not allow a user to execute a certain operation until all invalid constraints are fixed by the end user.

Persistence and specification of constraints should be done inside of Codi-framework. My module should only be able to work with the deep model, specify constraints and be able to verify it. So, my module should only serve as an engine to evaluate if a constraint is valid. Therefore, technology should be chosen that can meet the following aforementioned requirements. For this, I have to conduct multiple experiments.  
  
Current technologies in mind:

1. OCL using Eclipse (write a model transformation to convert codi structure into emf structure)
2. Use expression evaluation language (spel, jexl, ognl, mvel)
3. Consider writing your own ontology with OWL
4. Use SHACL as an alternative to OWL
5. Evrete as a rule engine

After the technology is chosen, one must think about a way how to help the end-user to write such constraints and implement them either in your module and embed it in codi or implement it in codi directly.

~~Apart from it, lambda updates can be considered or ENUM constraints should be considered (in this case the specification of ENUM should be added to codi)~~

~~What types of constraints? Attribute, association constraints during instantiation. Constraints on a model level (certain operations should be restricted on a model level if some constraints are defined) E.g., Car must have an attribute built-number if we want to have some other attribute or association.~~

# Meetup 10.10.2022

Key points that should be noted.

Nice concepts regarding SHACL and end-user interface. You could specify meaningful function blocks and then just apply them the question is whether it is possible to combine them.

How attributes and associations should be defined? My current approach is okay, in my reasoning scope I **MUST** expect that attributes or associations are unique. Karl says: “It looks pretty good for a prototype; the idea is nice to map this”.

Really nice to have: one that that I have done. How to bring my prototype to the domain of user-programming? Provide a set of predefined functions plus to have an ability to add those function blocks at runtime. – **END-USER PROGRAMMING.**

**EVALUATION IS IMPORTANT ALSO –** define use-cases from the simplest ones to the most complex ones and compare this set of constraints find theoretical solutions for each technology and compare them. **Look at related work for end-user constraint programming of runtime models.** Also, technologies for constraint specification and reasoning could be compared by me. – THIS IS EVALUATION REGARDING UI.

EVALUATION FOR CONSTRAINTS SHOULD BE DONE IN THE FOLLOWING WAY: implement 2 prototypes, one that I have already and the second one using e.g., Eclipse EMF and then compare them. Also, the chair is interested in how all those constraints can be applicable to other models. E.g. if the model is a graph then the RDF approach is much stronger because it operates on the graphs and OCL, EMF approach is weaker because it operates on Ecore, so find some comparison points with the focus on runtime models and how easily it is for users to specify functions and runtime to combine blocks. You could also claim that ui is independent of technology which is doing the reasoning and go with that.

Task description steps:

* Introduce the concept of end-user driven constraint modelling regarding the models at runtime and why expressive and flexible specification constraints are required – why should users model constraints at runtime?
* Conduct analysis for background chapter: models at runtime, multilevel models or other types of runtime models, constraint languages – I should focus on it.
* Analise what types of constraints, query or modelling languages are suitable to formulate useful restrictions on variant – aware entity models -> comparison and survey-based evaluation. Variant – aware -> be aware that e.g. if you say that a project can be related only to another project with a valid title. But the project you associate with has no longer a title, like what do you do – you can let it fail softly, don’t evaluate, etc. – it should be part of evaluation! Or if a project has a title and there is a constraint on that and then at some time this title attribute is removed. What to do with the constraint?
* Evaluate technical properties, how easily it can be implemented or how fast it is
* How the queries can be constructed by the end-user.
* Choose two promising techniques, implement them in form of the framework extension and compare them and how they can be applied at runtime. And then say that one is better for modicio but maybe another one would be better for another suit-case.
* If there are some queries that can’t be realize in the prototype, you should write about what should be done in order to realize those constraints in the future.
* Mark some constraints more important than others.

**Citations:**

Regarding citations: I recommend to store each paper you look at somewhere together with some notes which paper is about what and also always copy the bibtex citation json to include in latex. Latex then actually only compiles those references at the and which are used in the text

Also, we cite somewhat relaxed as you can also see in my template.

1. If you use a special term which was introduced somewhere, then citation directly afterwards.
2. If you explain something that you read somewhere, then the citation comes at the end of the sentence or paragraph which is based on the source.
3. Direct 1 to 1 citations should be avoided. But if you require to directly cite a definition, you can wrap in in quote blocks (see overleaf doc). If you directly use an image as I did with the Type Object pattern, also directly add the citation.