Optimistic Versioning for Conflict-tolerant Collaborative Blended Modeling

FPVM workshop

July 4, 2022

Joeri Exelmans, Jakob Pietron, Alexander Raschke
Hans Vangheluwe, Matthias Thichy

Versioning in model-driven engineering

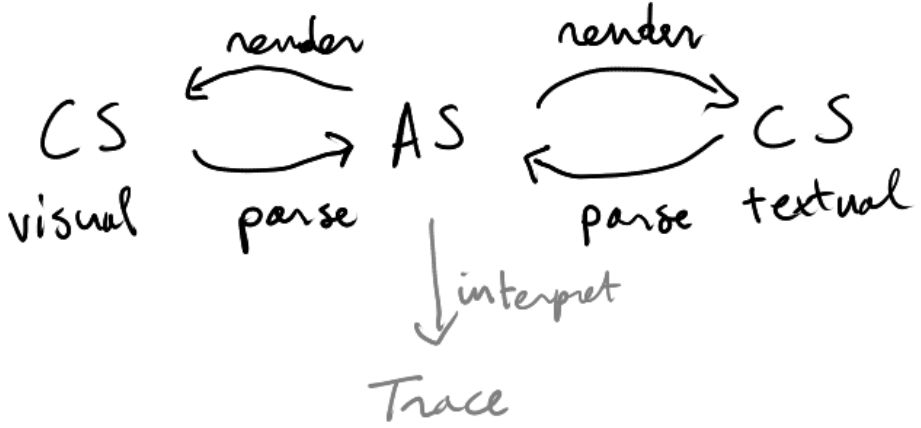
= *Central piece of technology* in any kind of collaborative work

- Text-based versioning (e.g. Git) insufficient
  - intended to take snapshots of **textual concrete syntax** (for *code*)
  - for models, can only take *snapshots* of serializations (e.g. XML) of models
  - introduces lots of accidental complexity
- Research in **model versioning** addresses this by recording history, merging, detecting conflicts at the level of **abstract syntax**
  - a 1:1 mapping between concrete and abstract syntax elements is assumed (i.e. concrete syntax == abstract syntax “with icons”)
  - very little “concrete syntax freedom”
  - cannot support “Blended modeling” (next slide…)

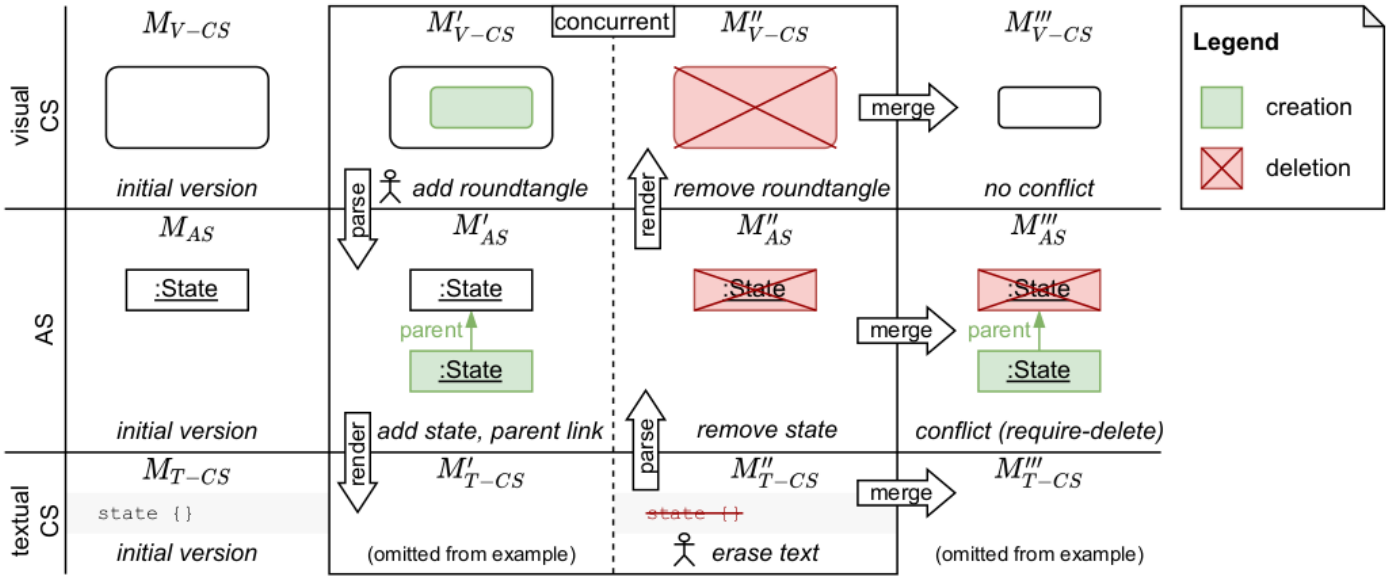
Blended modeling

= Ability to *edit* a model (= abstract syntax) *through different representations* (= concrete syntax)

- **Usability++** User can choose the representation that is most efficient for the current task
  - e.g. understanding links between elements -> observe visual layout
  - e.g. renaming an element -> find/replace in textual CS
- Technical challenge: bi-directional synchronization between CSs and AS



Running example: Blended modeling × Concurrent edits



- Complexities:
  - Non-trivial CS <-> AS mappings
  - Bi-directional change propagation (CS <-> AS <-> CS)
  - Concurrent edit operations
- Minimal, but representative of real-world scenarios

Blended modeling × Concurrent edits

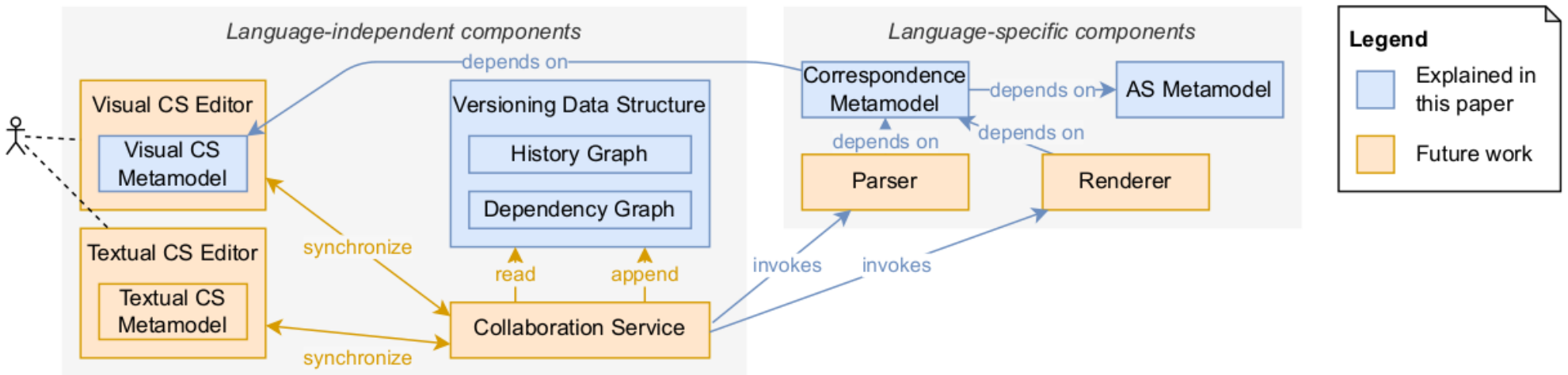
Two sources of complexity:

- Concurrency (branching, merging, detecting & resolving conflicts)
- Bi-directional change propagation (CS <-> AS <-> CS)

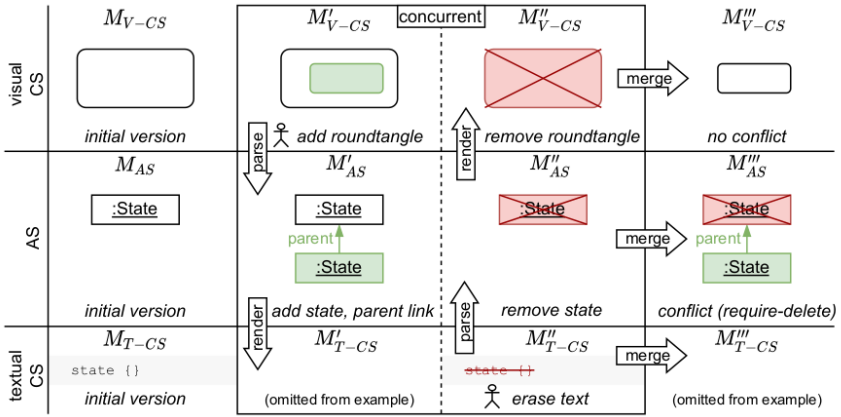
Want to keep these complexities *orthogonal*
... and we show that we can :)

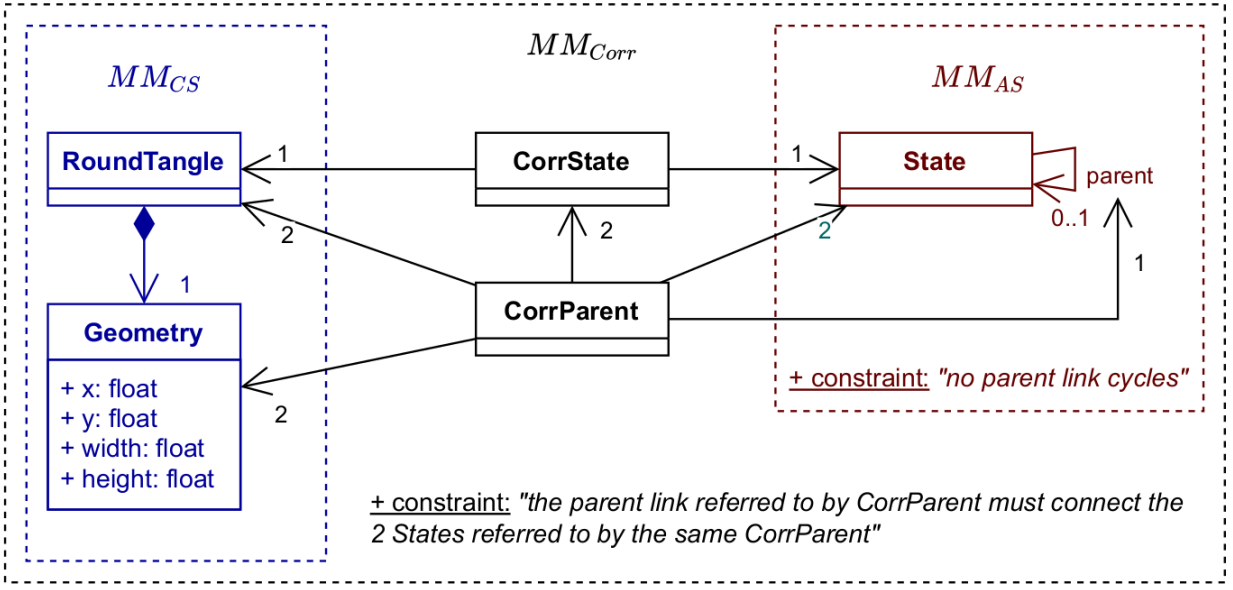
Big picture of solution

- CS and AS are **independently versioned**, and each have their own metamodel
  - Inspired by: Y. Van Tendeloo: **Unifying Model- and Screen Sharing** (2018)
  - Running example:
    - CS (visual) metamodel: (vector) drawings
    - AS metamodel: Statecharts (simplified)
- Record correspondence links between CS and AS elements in **correspondence model**
  - Idea taken from Triple Graph Grammars
  - Correspondence model is also **versioned**

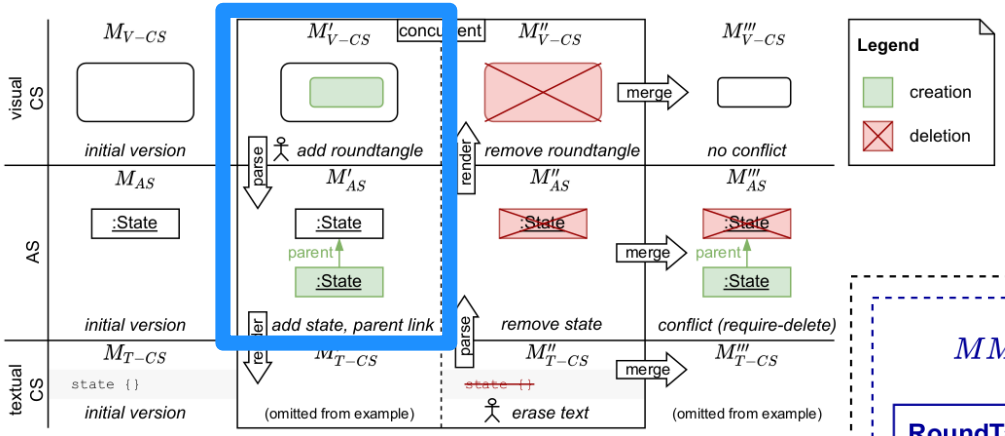


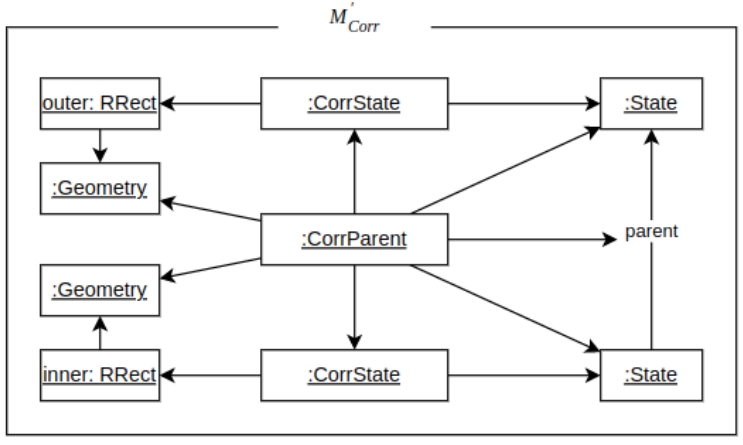
Running example: metamodels of CS, AS, corr

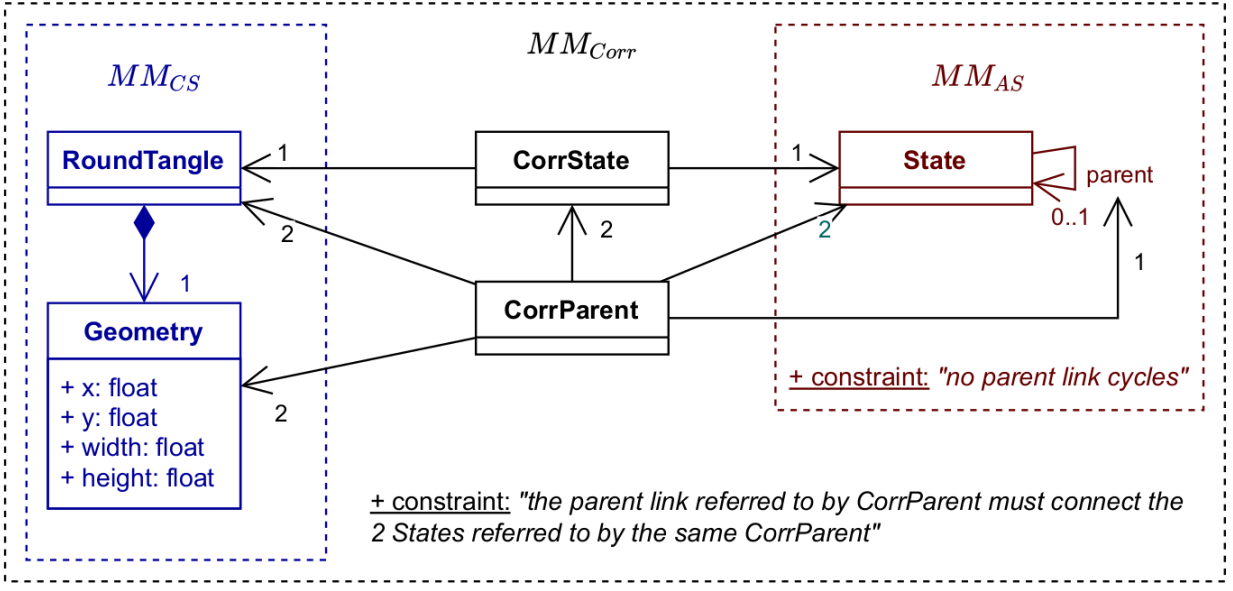




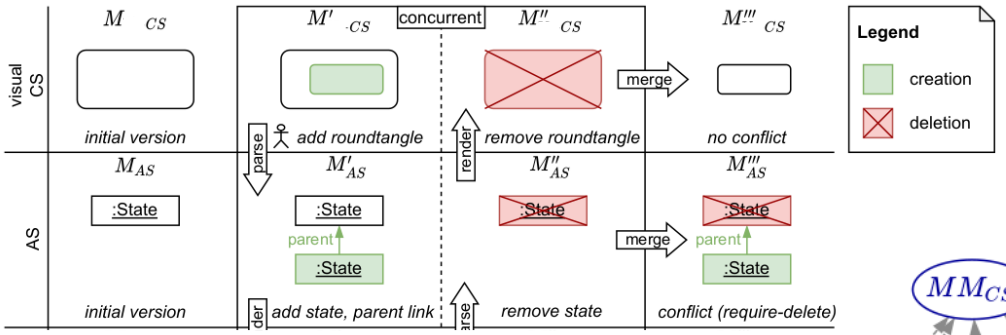
Running example: metamodels of CS, AS, corr

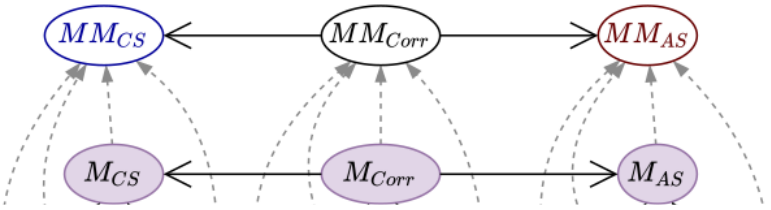


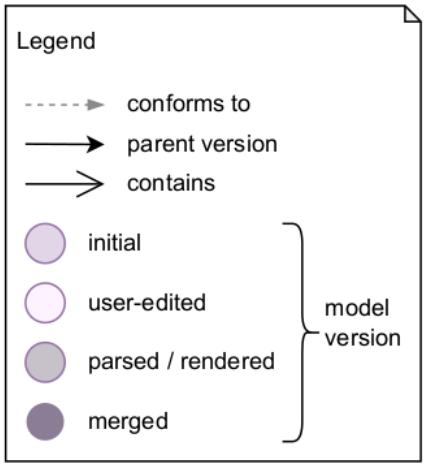




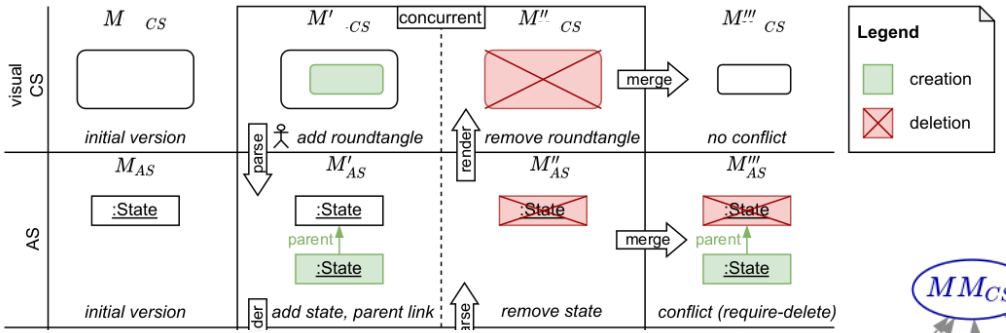
Running example: Evolving CS, Corr, AS

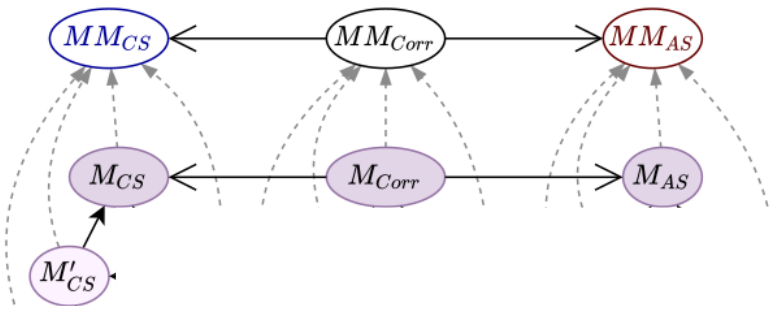


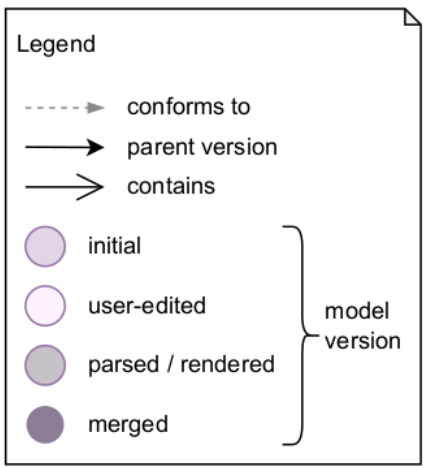




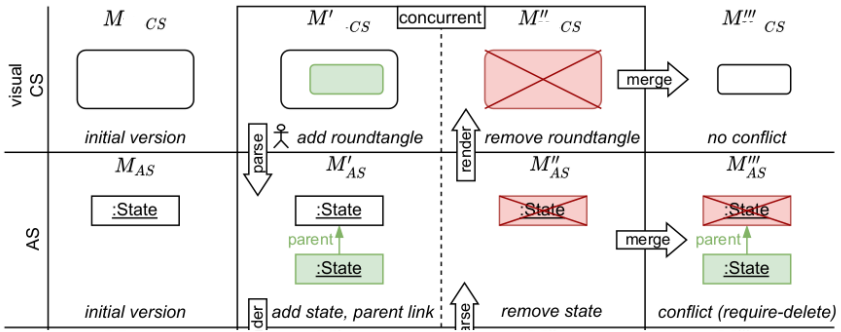
Running example: Evolving CS, Corr, AS

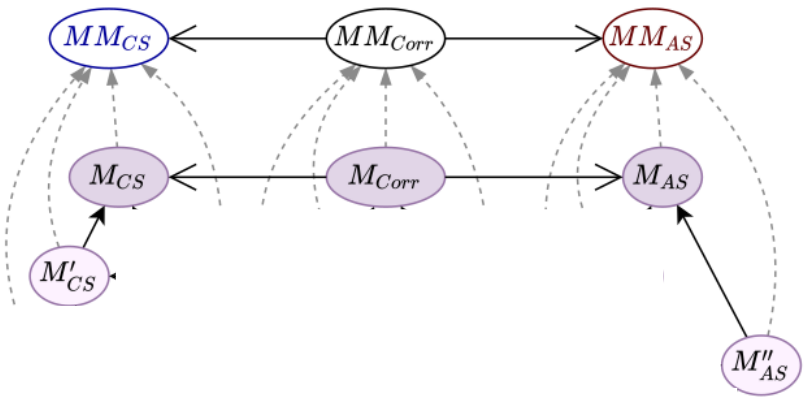


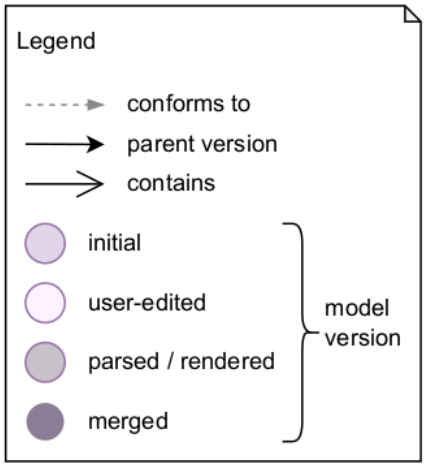




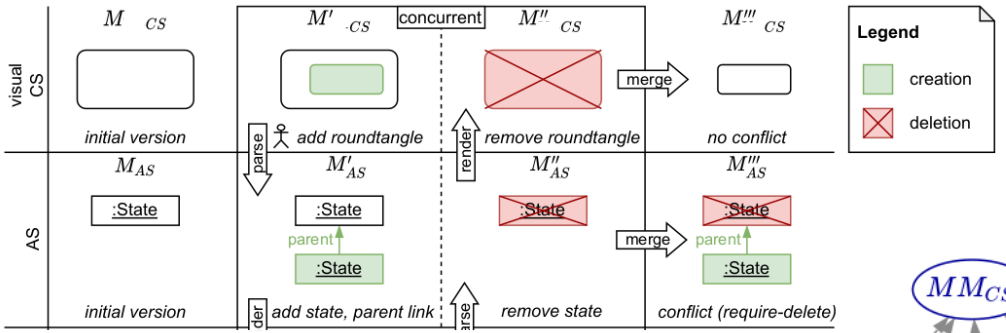
Running example: Evolving CS, Corr, AS

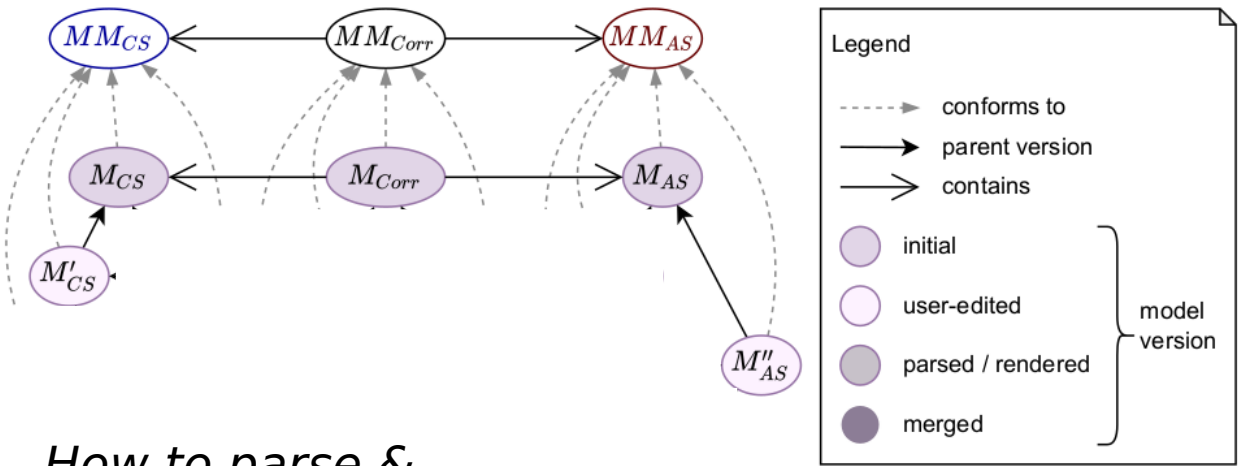






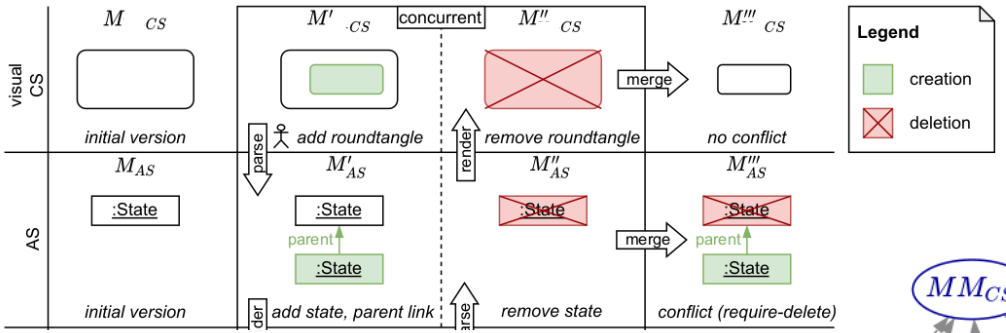
Running example: Evolving CS, Corr, AS

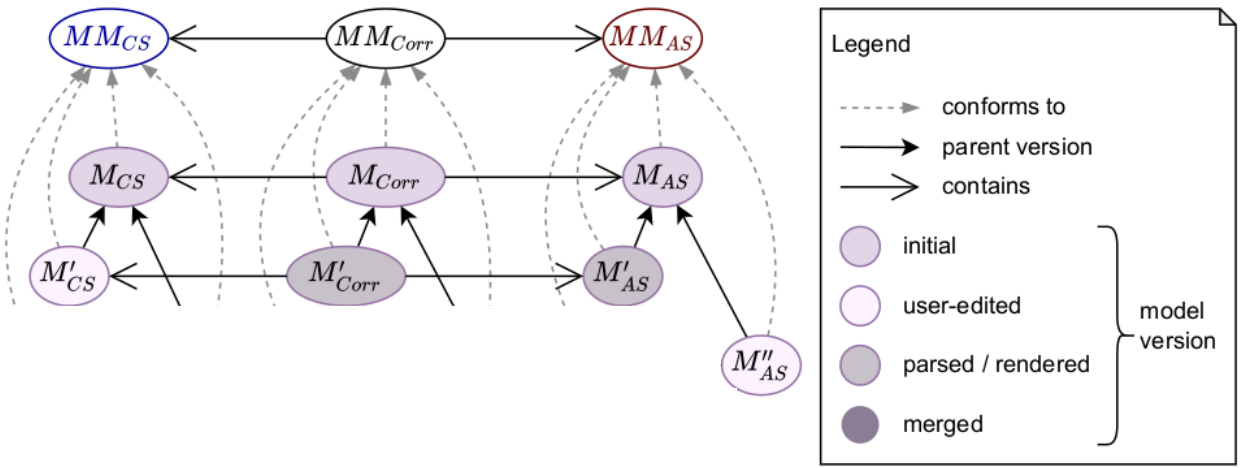




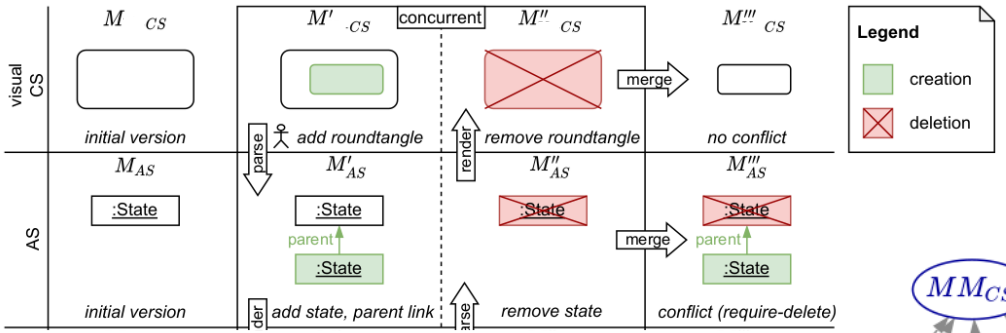
How to parse & render?

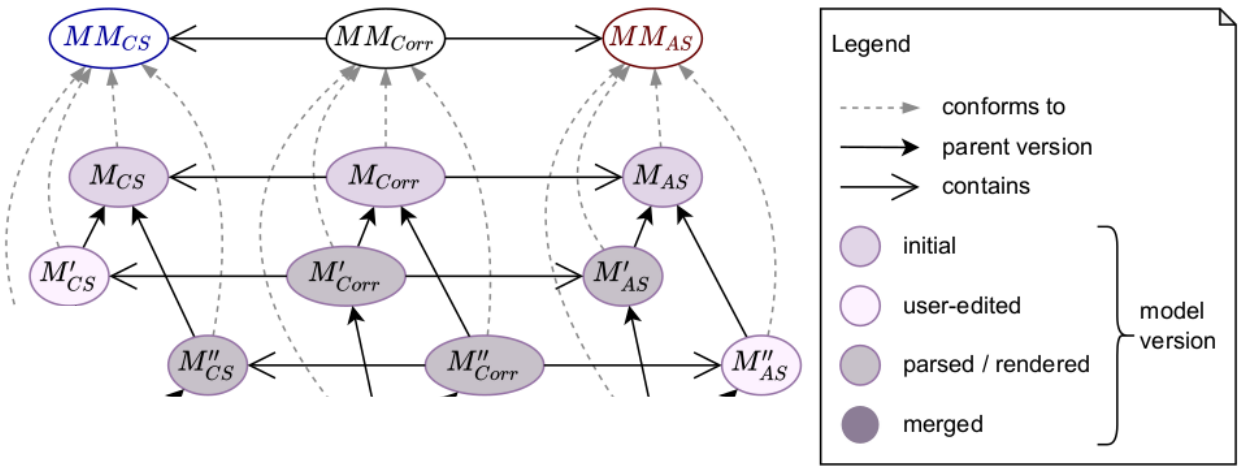
Running example: Evolving CS, Corr, AS



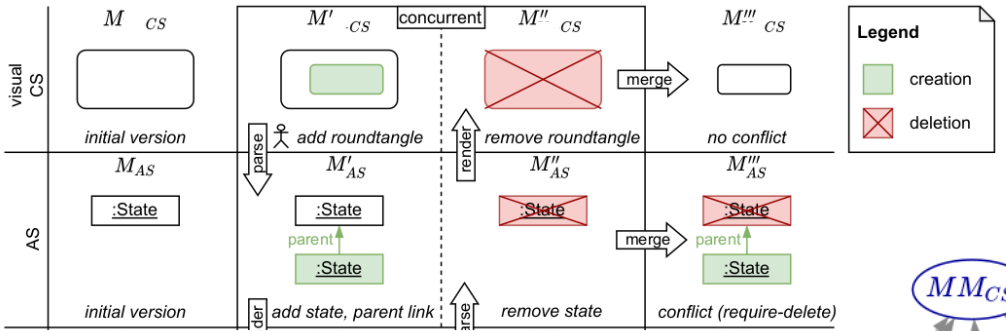


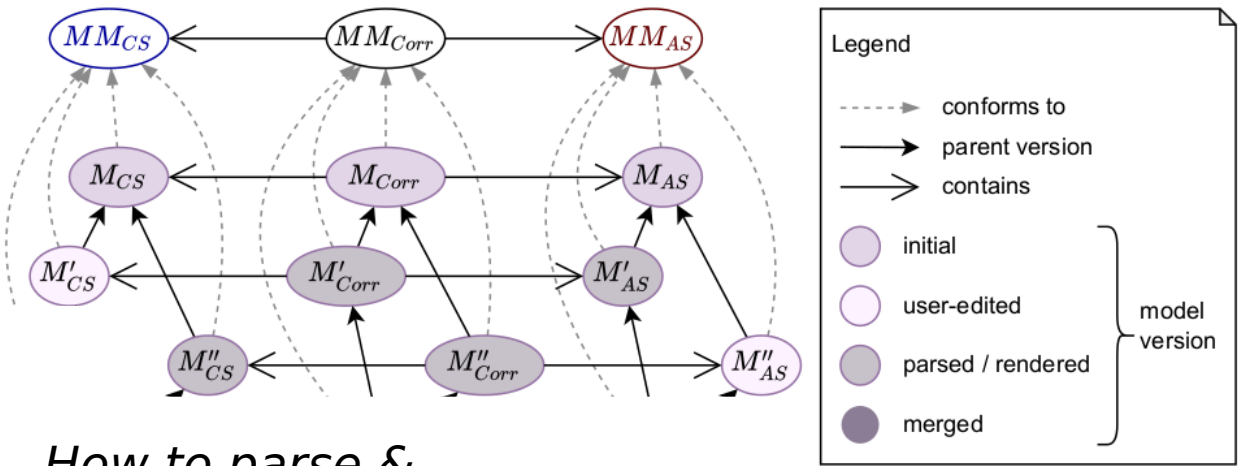
Running example: Evolving CS, Corr, AS





Running example: Evolving CS, Corr, AS

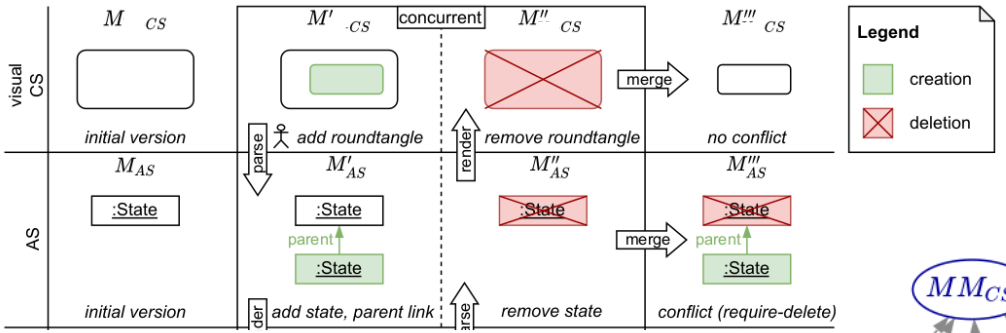


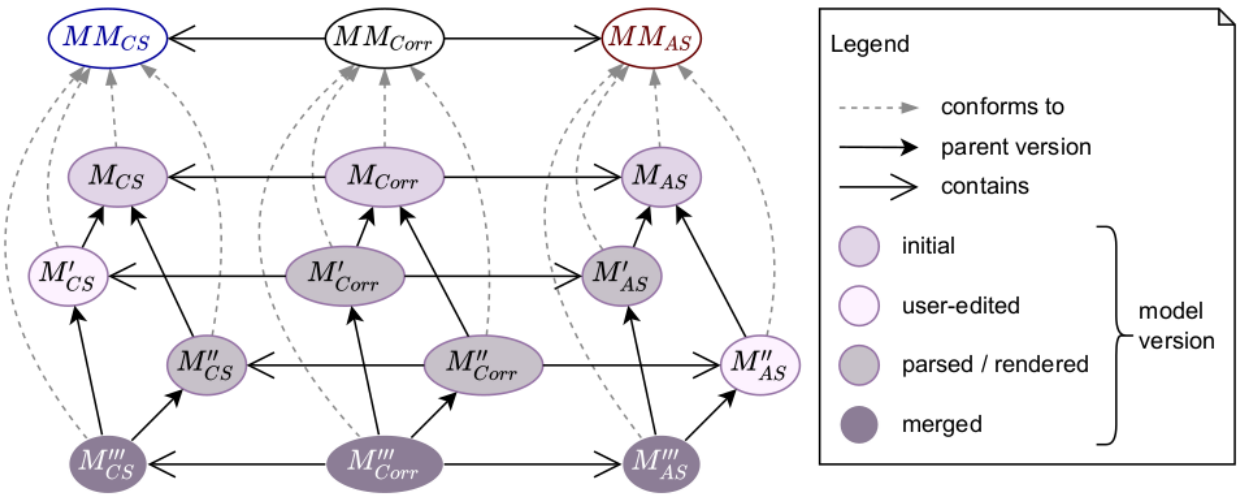


How to parse & render?

=> just create new model versions!

Running example: Evolving CS, Corr, AS





Still unclear at this point...

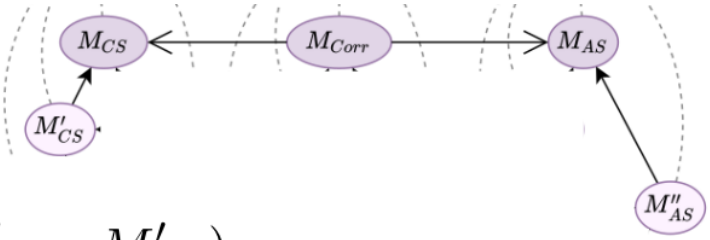
- How do we **parse & render**?
- What exactly is **recorded** in history?
- How do we **merge** concurrent versions and **resolve conflicts**?

Parsing and rendering must happen **incrementally**

Meaning: a change to CS must only cause a corresponding change to AS (instead of generating a new AS model), and vice versa.

- Reasons:
  - for performance
  - for layout continuity
  - to deal with concurrency
- Fits nicely with **operation-based versioning** (= storing only the changes to the model, as opposed to storing snapshots)
  - in our case, we store simple CRUD operations on model elements (i.e. the *effect* of edit operations and change propagations)
  - a conflict is a simple overlap (update-update, update-delete, delete-require)

Parsing and rendering **interfaces**



$$parse(M_{Corr}, d(M_{CS_V}, M'_{CS_V})) = (M'_{Corr}, M'_{AS})$$

$$render(M_{Corr}, d(M_{AS}, M''_{AS}), *) = (M''_{Corr}, M''_{CS_V})$$

where d is the difference (i.e. added, removed elements) between two model versions

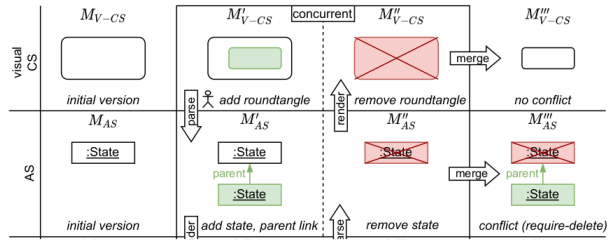
- Parsing:
  - always deterministic
- Rendering:
  - often not deterministic
    - visual CS: what layout to use?
    - textual CS: in what order to put declarations?
  - coping mechanisms:
    - postponing rendering (i.e. until CS is opened/requested)
    - support **human interaction**
      - additional tweakable parameter(s) in render function (e.g. random seed, heuristic to optimize, …)
      - upon human disagreement (“how should this be rendered”), a conflict at the level of CS occurs.

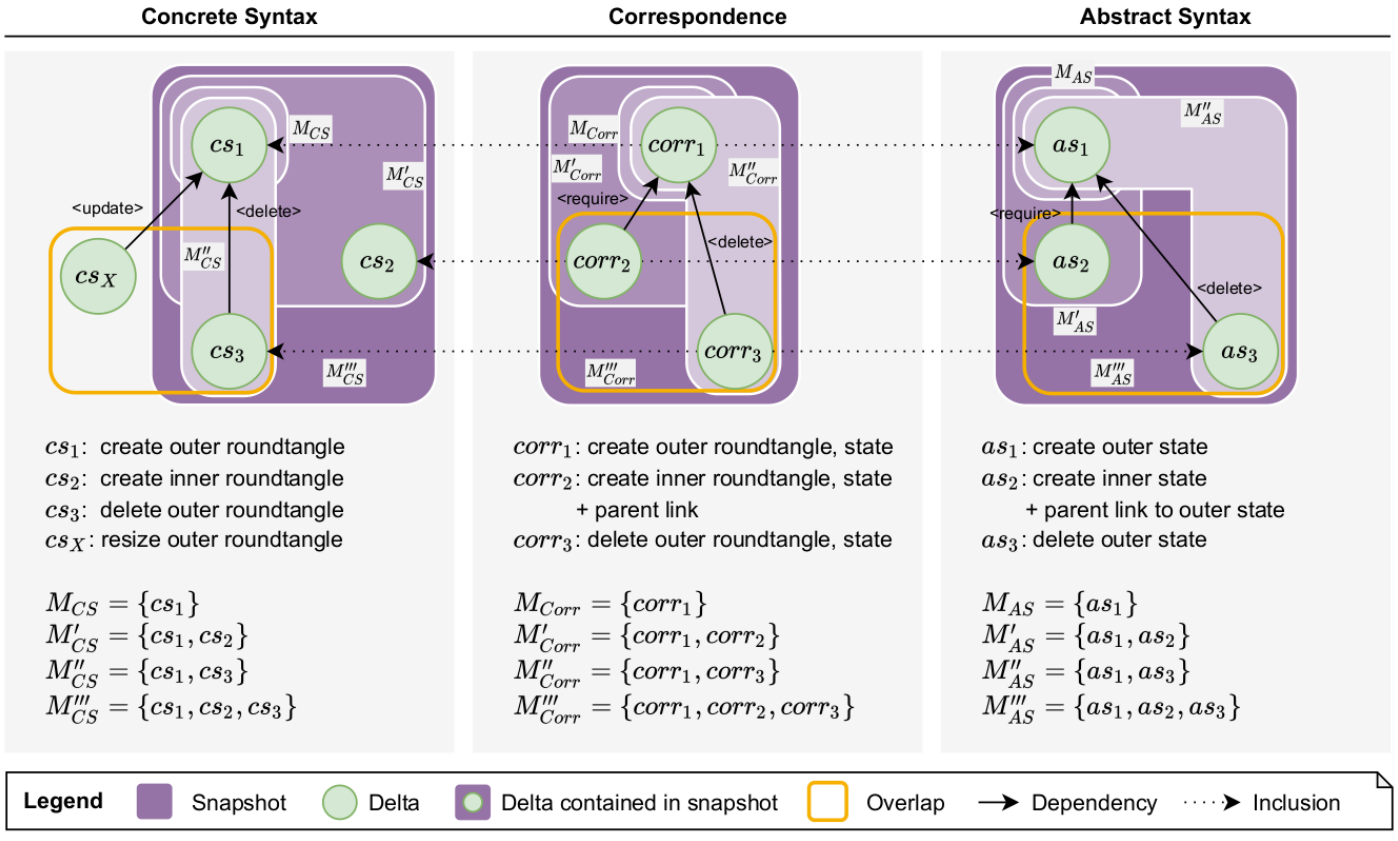
Persisting history

- Two data structures:
  - **Delta graph**: set of *deltas* and dependencies between them
    - delta = set of CRUD operations on model elements (effect of user edit or change propagation)
    - three types of dependencies: (1) update (2) delete (3) require
    - a *left-closed* set of deltas == a snapshot

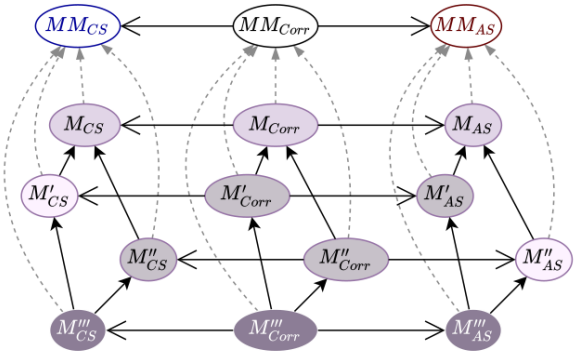
      meaning: for every delta in the set, its dependencies are also in the set
    - purpose of delta graph:
      - efficiently **detect conflicts**
        - a conflict is an overlapping (concurrent) dependency: update/update, update/delete, delete/require
      - always have valid snapshots (e.g. when undoing, resolving conflicts)
        - by enforcing left-closedness
    - dependencies form a **directed acyclic graph**
    - append-only!

Running example: Delta graph





Persisting history (2)



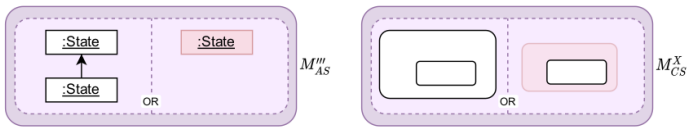
- Two data structures:
  - **Delta graph**
  - **History graph**: expresses **order** between snapshots
    - order expressed by “parent” links (identical semantics to parent links in Git)
    - parent links for a **directed acyclic graph**
    - append-only!
    - snapshots can be merged by taking the union of their parent snapshots
      guaranteed to also be *left-closed*
    - we **allow conflicting deltas** in snapshots

Persisting conflicting snapshots

Why are conflicts not just temporary, in-memory phenomena during the merge process, but instead stored forever?

- Can record the steps taken in the resolution of a conflict (as a sequence of snapshots)
  - what was the conflict?
  - how was it is resolved?
  - when was it resolved?

  = *valuable information!*
- Conflicts do not have to be immediately resolved
  - Editor should **visualize conflicts**, though



  - Modeler can continue working on non-conflicting parts of the model
- Conflict resolution happens by creating a follow-up snapshot that no longer contains a conflict (e.g. by excluding one of the conflicting deltas)

Conclusion

- Presented an approach for **optimistic versioning** that supports **blended modeling** and **reuse of concrete syntax metamodels** and **editors**
- “Just record everything” (and impose nothing) philosophy:
  - record CS, AS, correspondence
  - record changes
  - record snapshots
  - record conflicts

  Sandbox, possibly for empirical study

Future work

- Investigating types of **inconsistencies** that can occur, at level of:
  - CS
  - AS
  - Correspondence
- Having multiple intermediate layers between CS and AS
  - Motivation: **reuse** of certain CS features: e.g. geometric “insideness” is a feature of several visual languages
- Prototype web-based **implementation** of running example