Decentralized Operation-based Graph Versioning to support **Collaborative Blended Modeling** and **Live Modeling**

ENSTA Bretagne, Brest
21 September 2023

Joeri Exelmans

University of Antwerp

Overview

- **Part 1: Collaborative Blended Modeling**
  - Background
    - Versioning
      - “Classic” text-based versioning (git) and its limitations
      - Going beyond git
      - Versioning in Model-Driven Engineering
    - Blended Modeling
  - Research question
  - Running example
  - Solution (top→down)
- **Part 2: Collaborative Live Modeling**
  - Background: live modeling
  - Research question
  - Running example
  - Solution

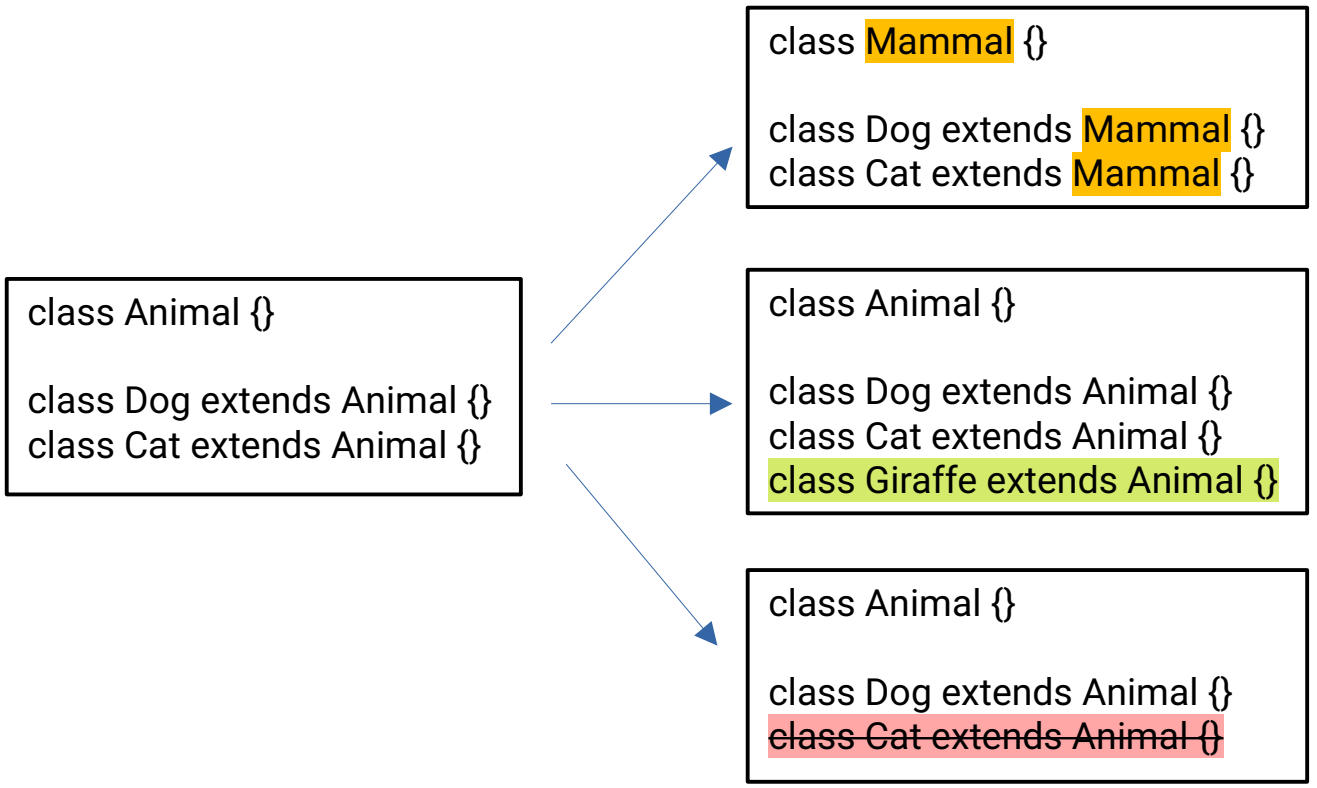
Part 1: Collaborative Blended Modeling

Versioning

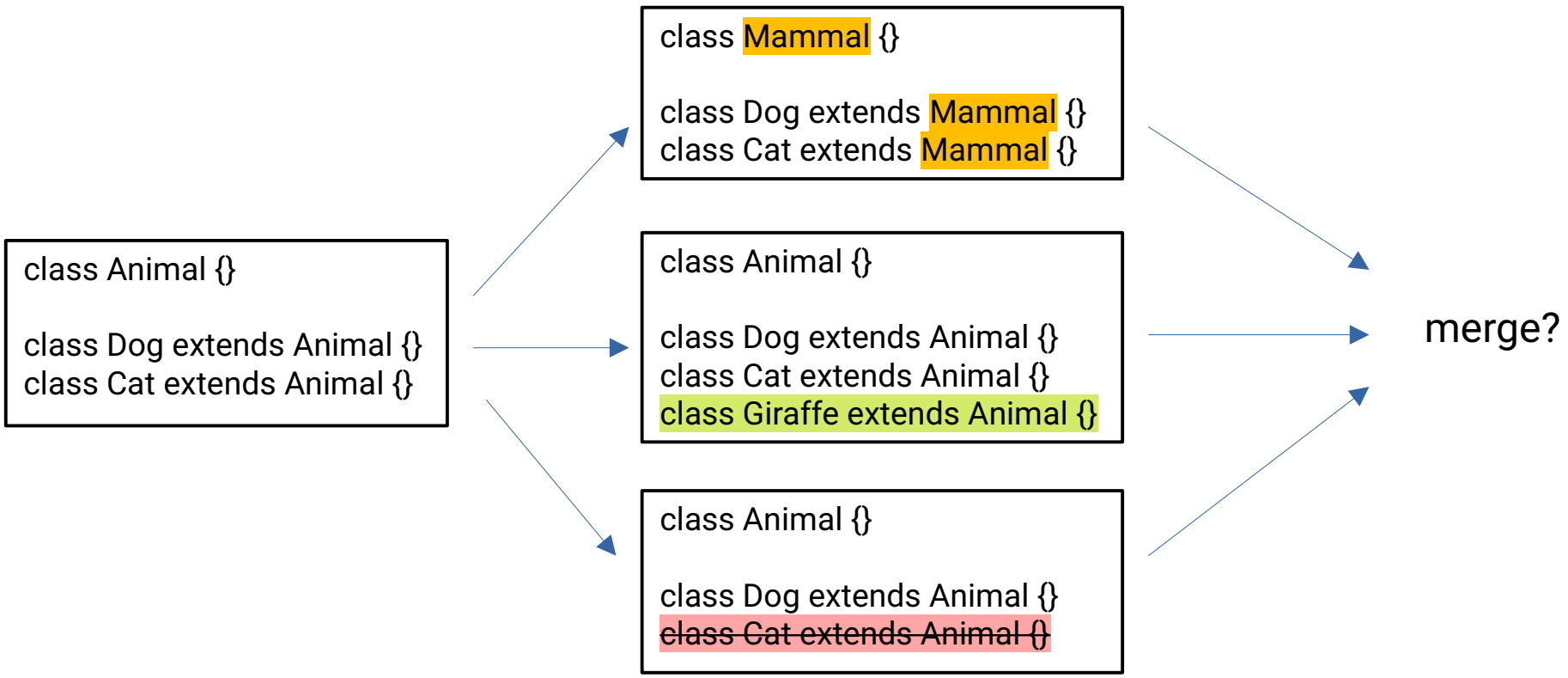
= ***Central piece of technology*** in any kind of collaborative work

- programming
- modeling

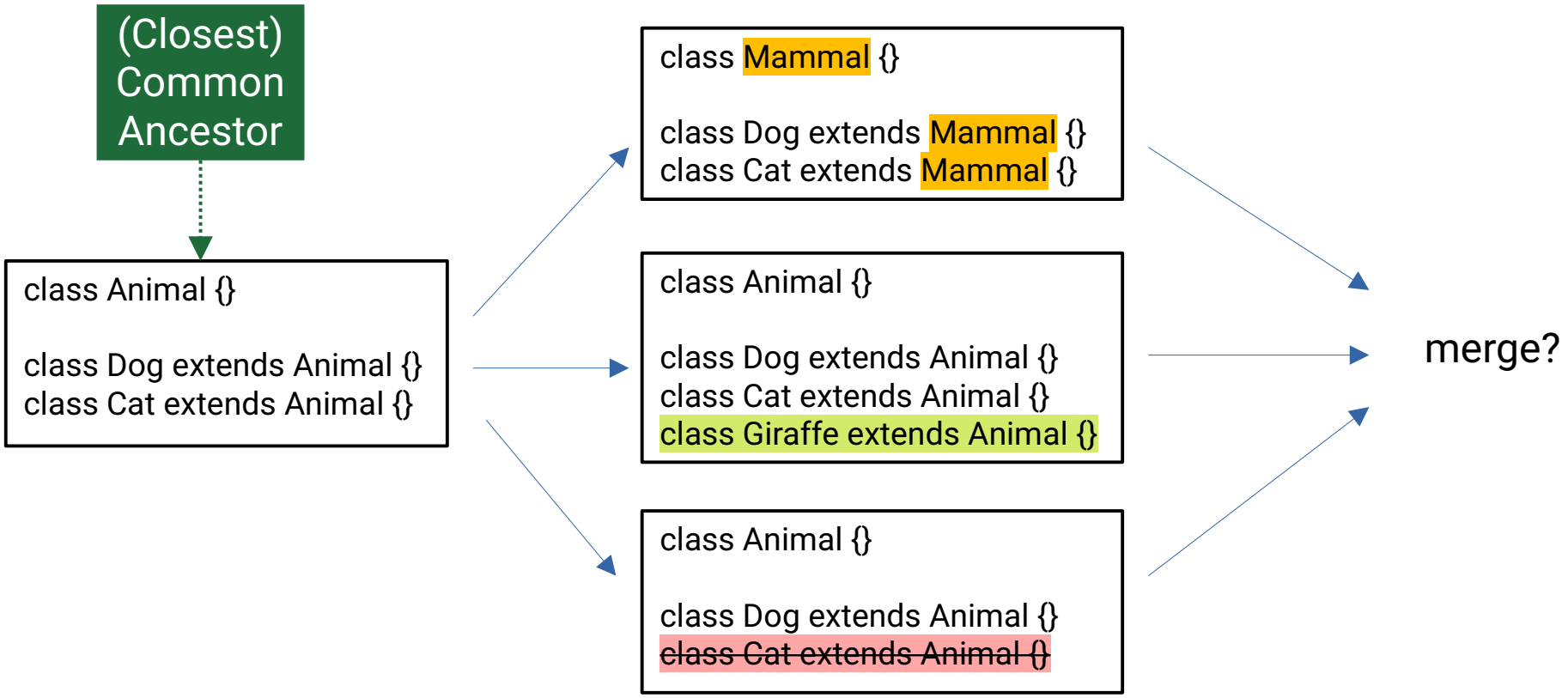
“Classic” text-based + snapshot-based versioning (e.g., git)



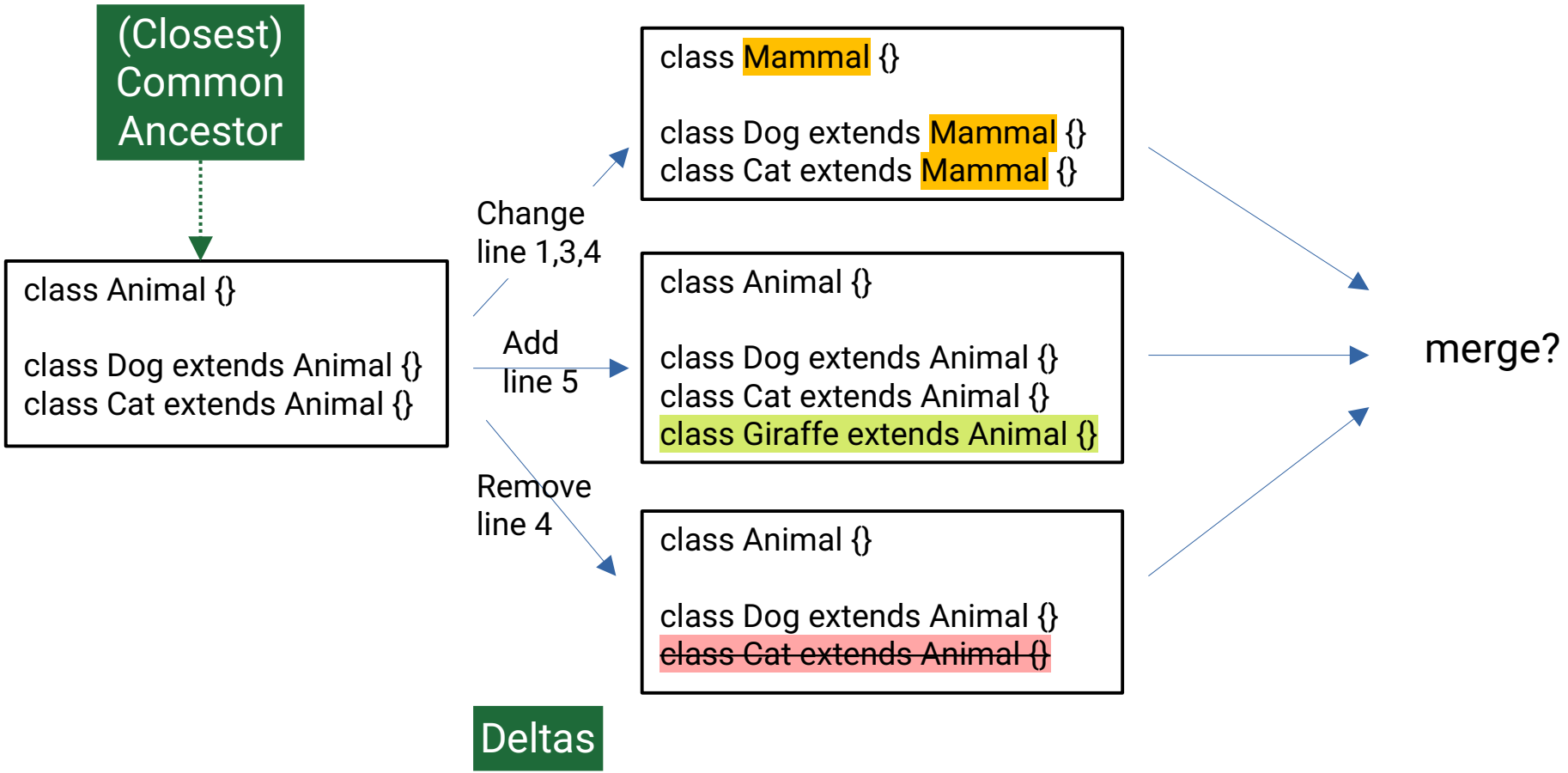
“Classic” text-based + snapshot-based versioning (e.g., git)



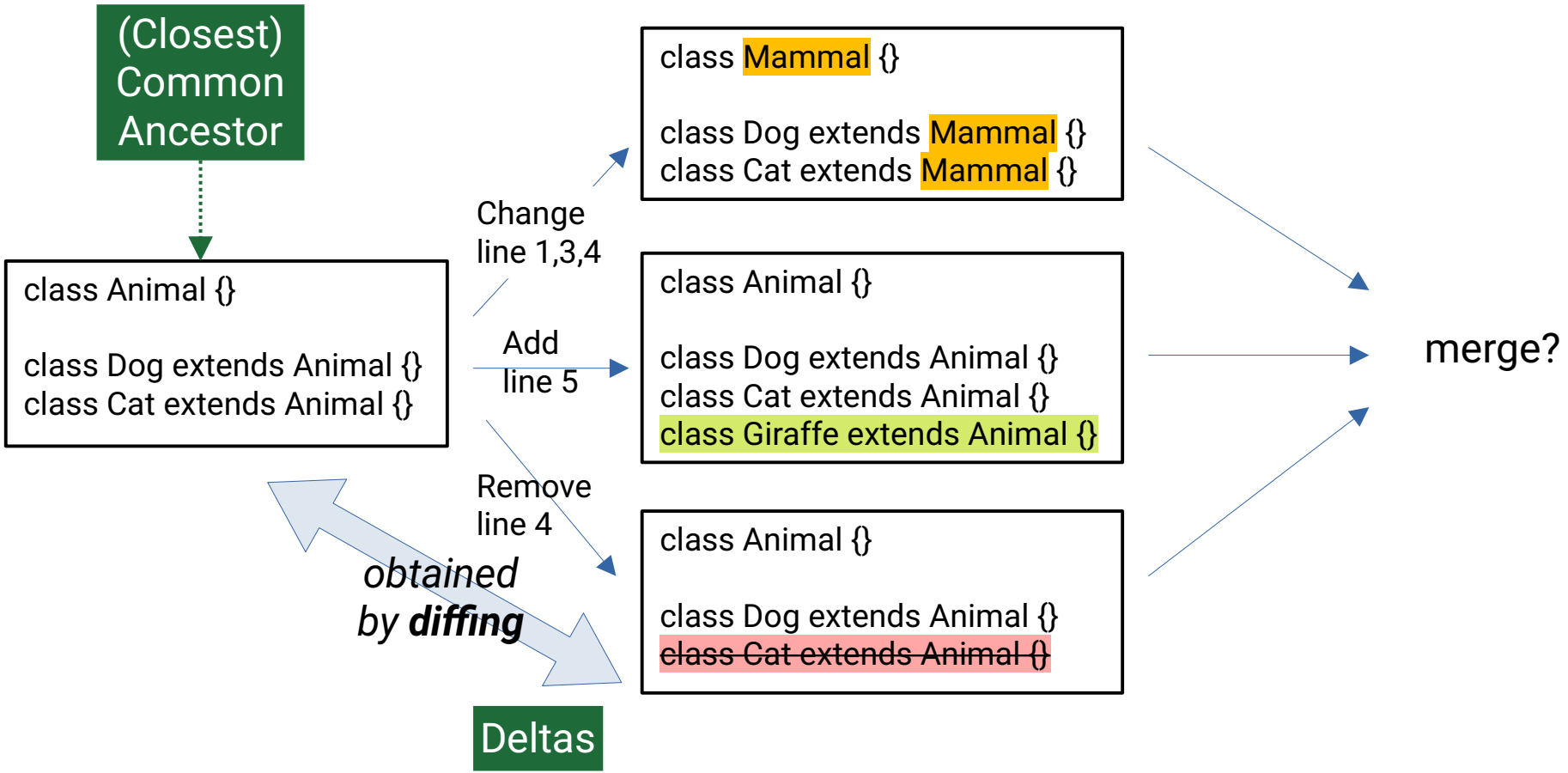
“Classic” text-based + snapshot-based versioning (e.g., git)



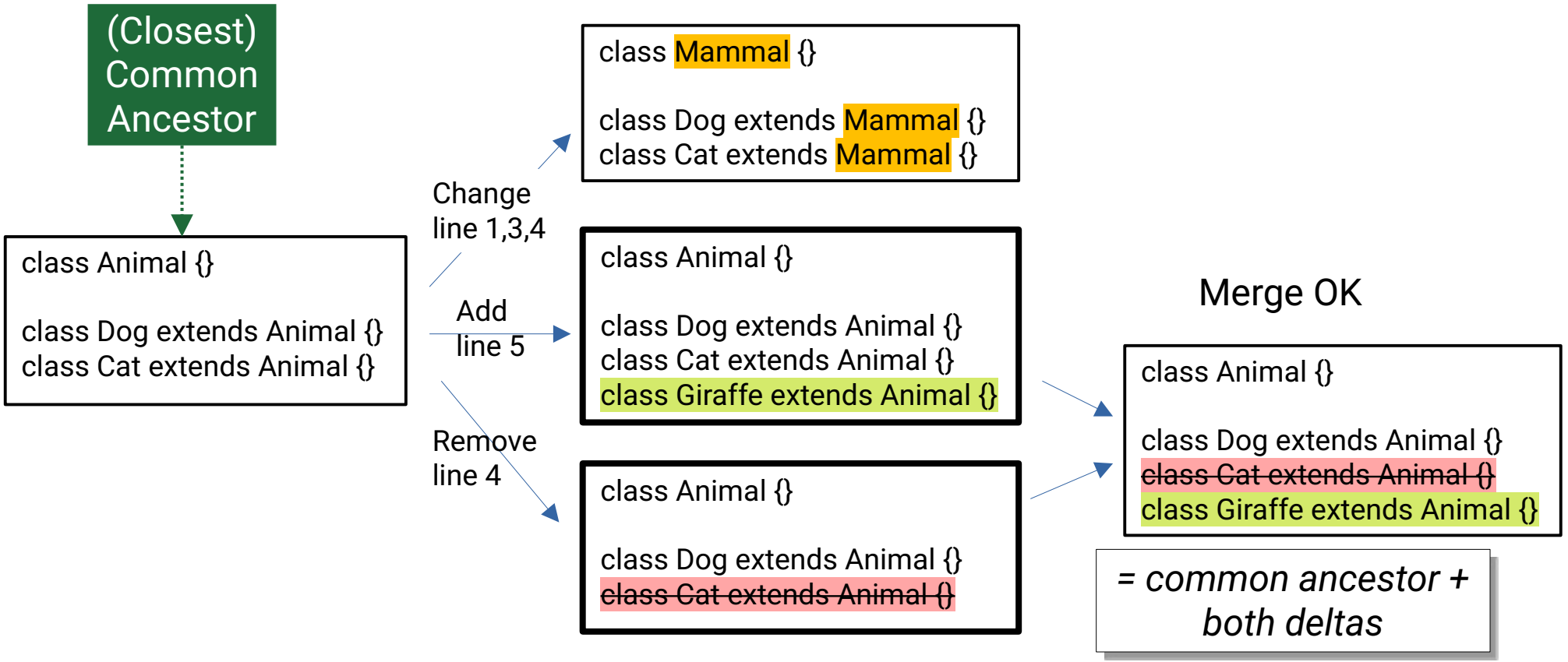
“Classic” text-based + snapshot-based versioning (e.g., git)



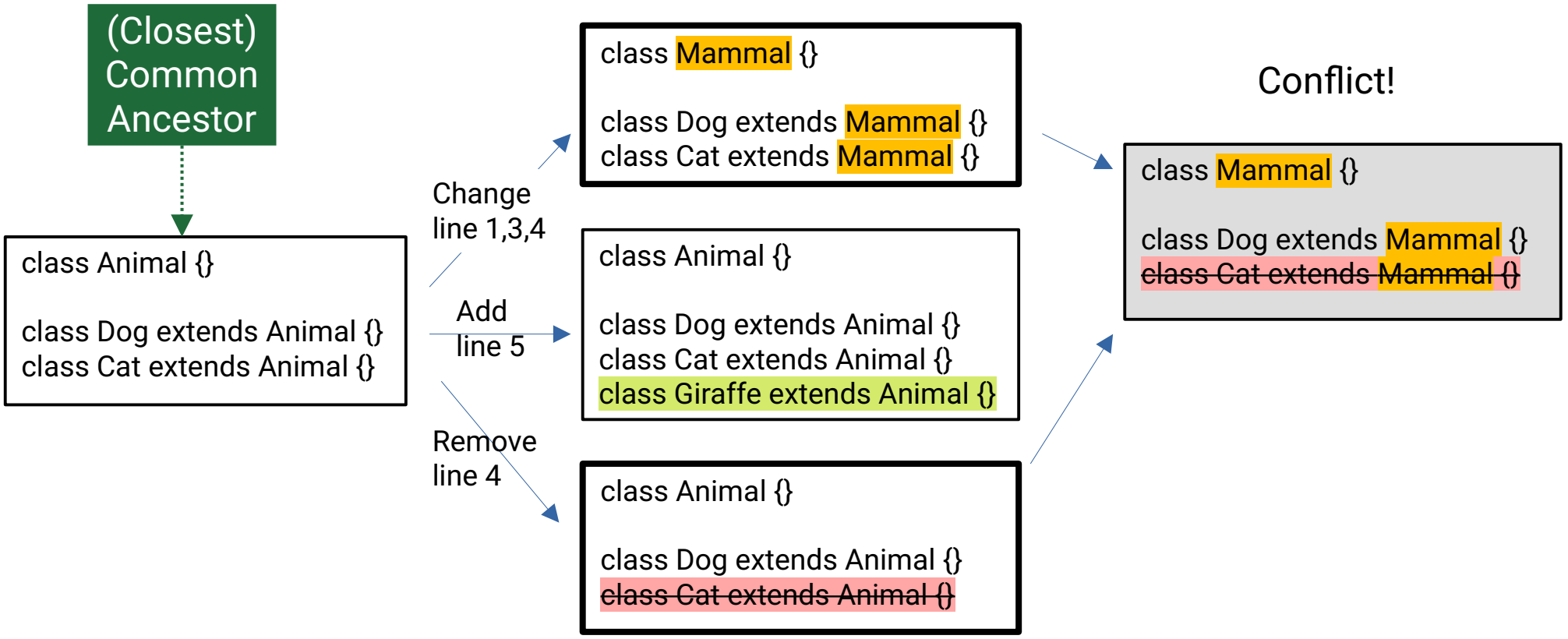
“Classic” text-based + snapshot-based versioning (e.g., git)



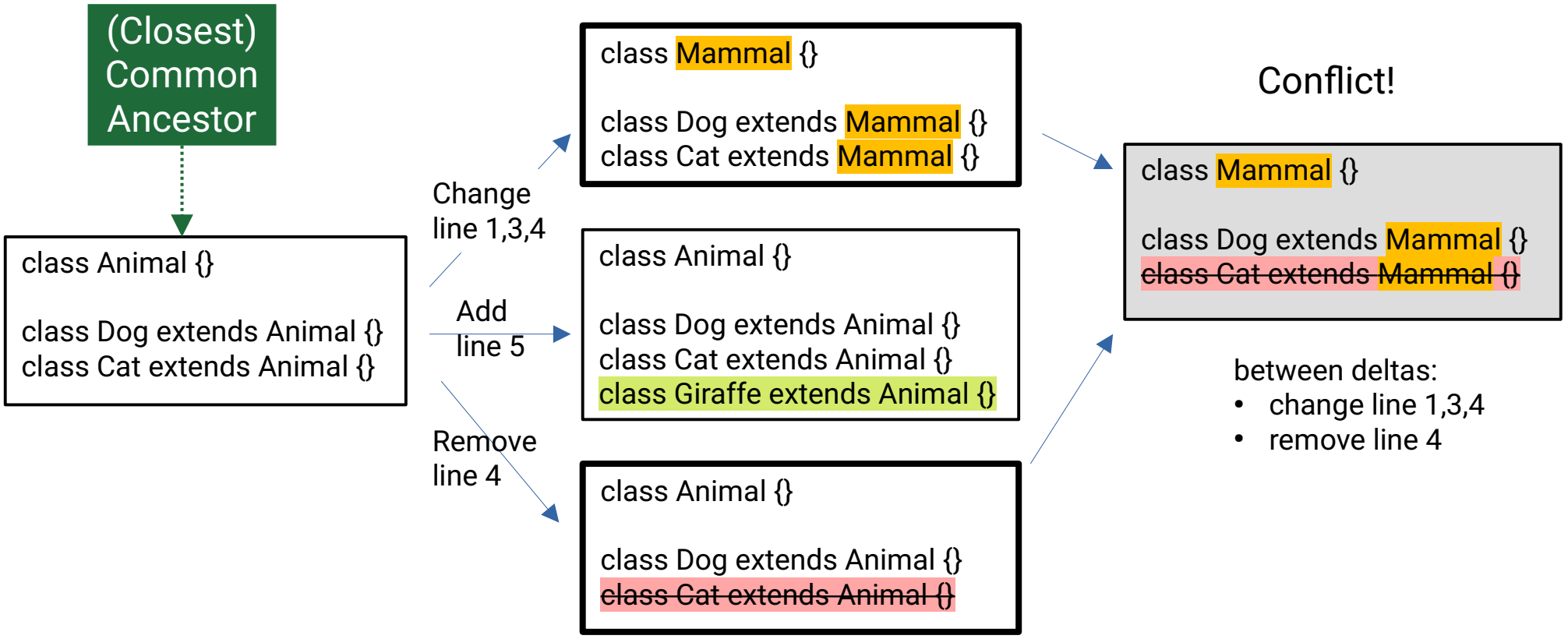
“Classic” text-based + snapshot-based versioning (e.g., git)



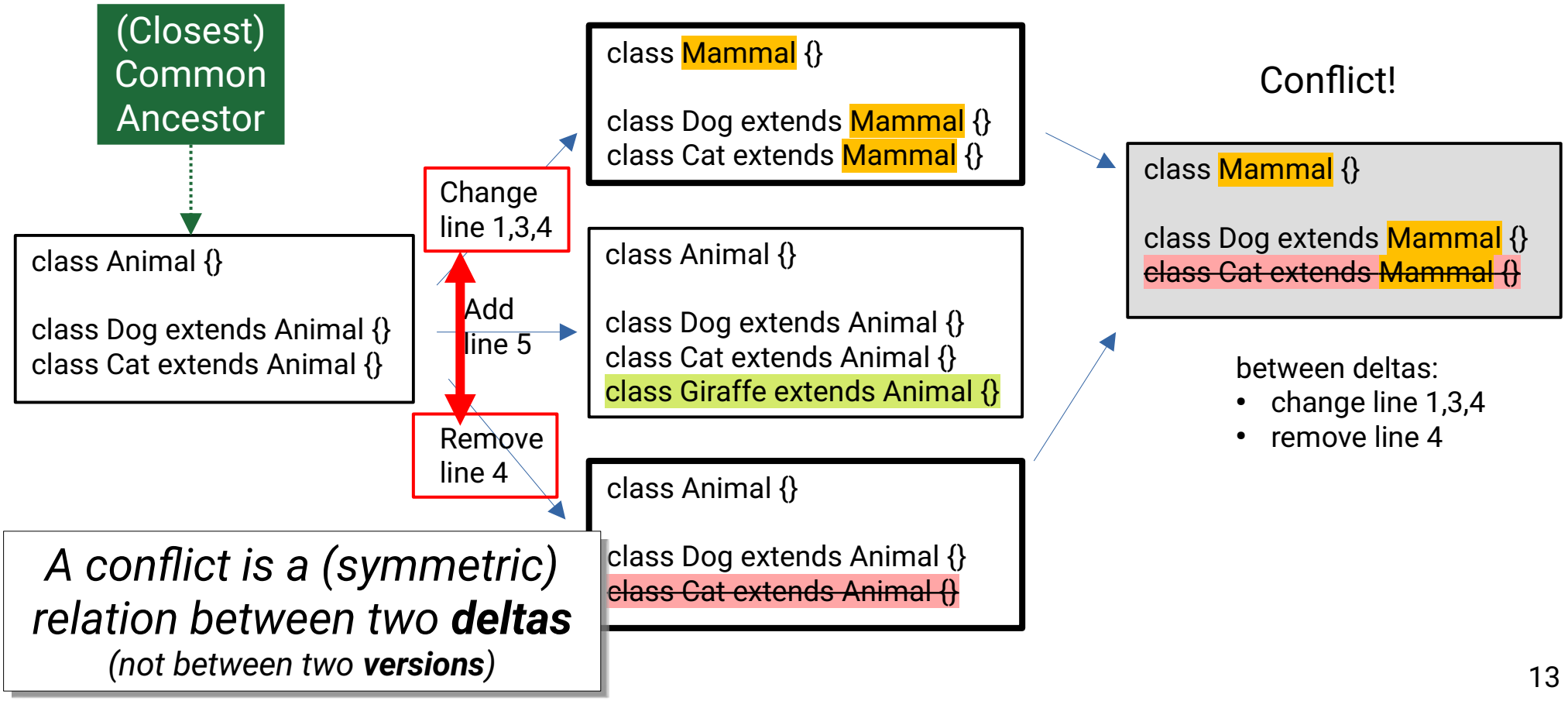
“Classic” text-based + snapshot-based versioning (e.g., git)



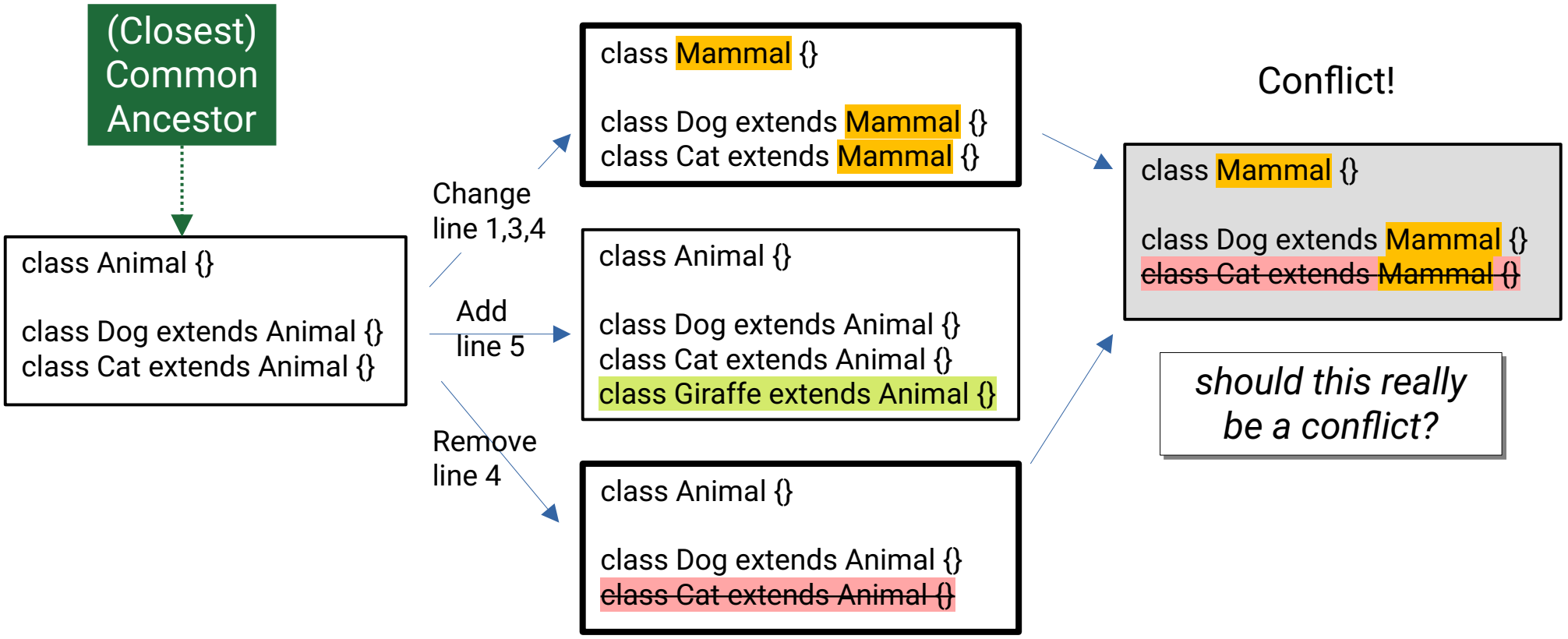
“Classic” text-based + snapshot-based versioning (e.g., git)



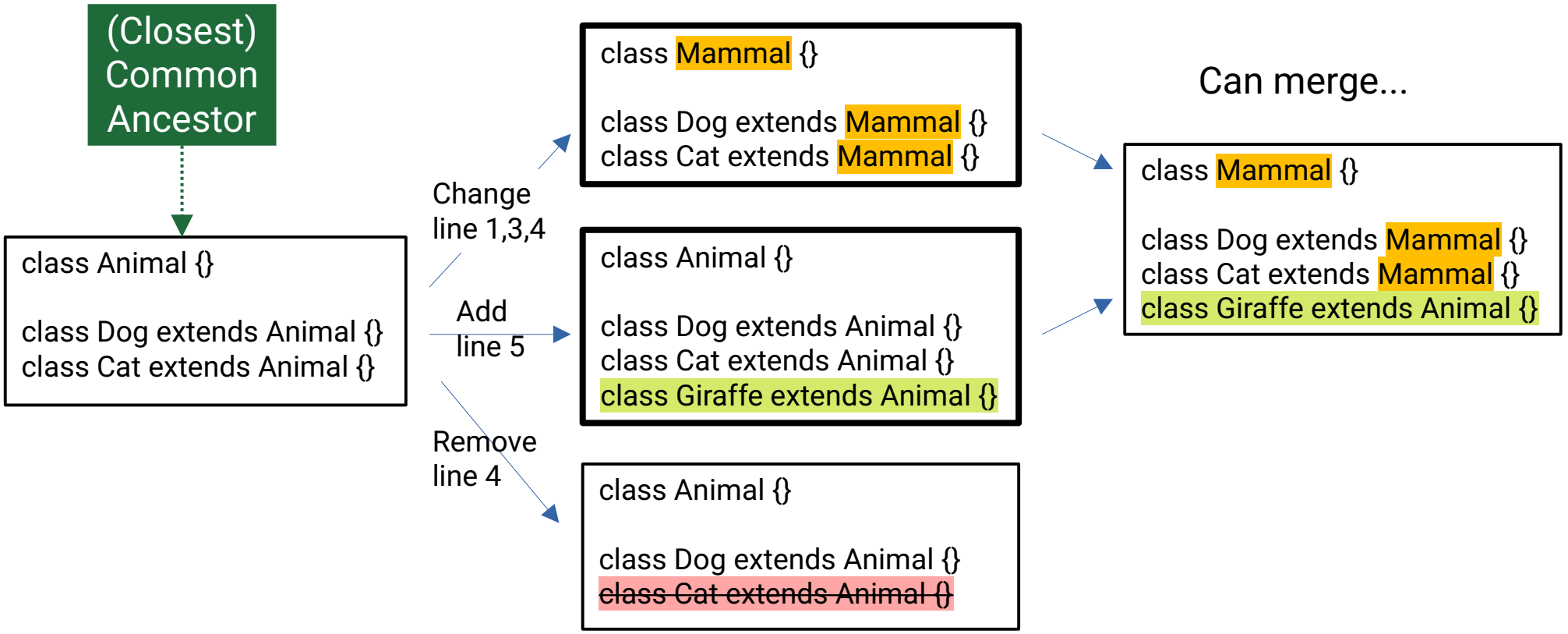
“Classic” text-based + snapshot-based versioning (e.g., git)



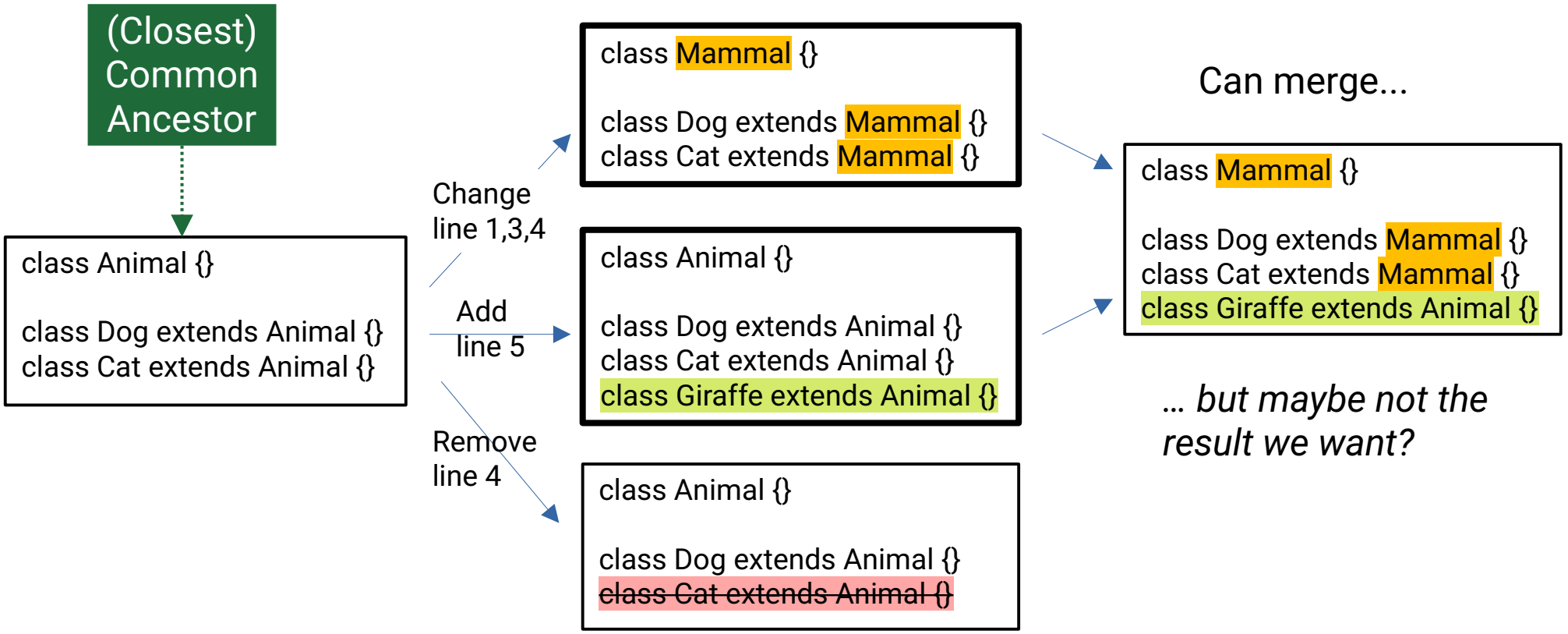
“Classic” text-based + snapshot-based versioning (e.g., git)

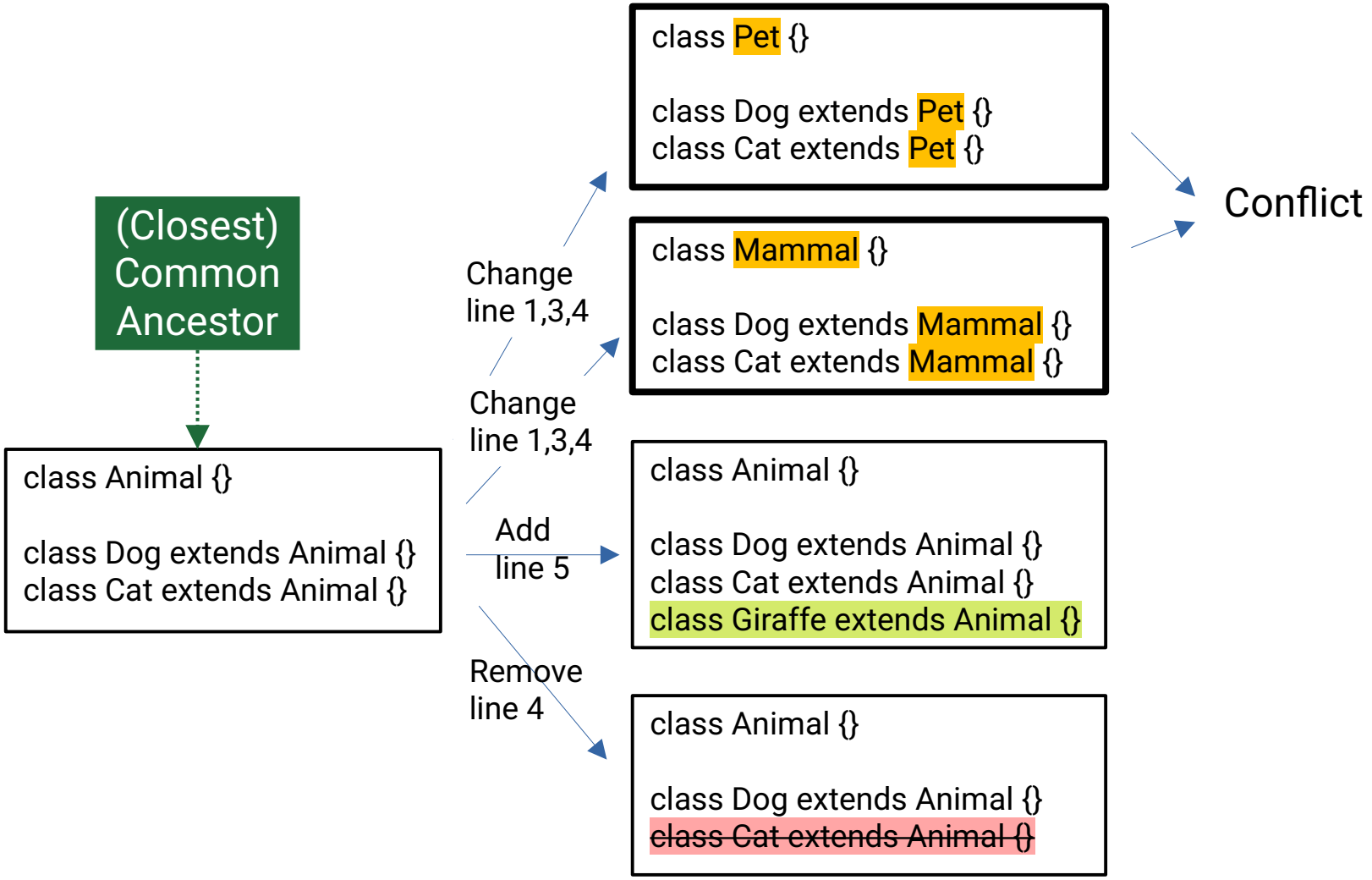


“Classic” text-based + snapshot-based versioning (e.g., git)



“Classic” text-based + snapshot-based versioning (e.g., git)



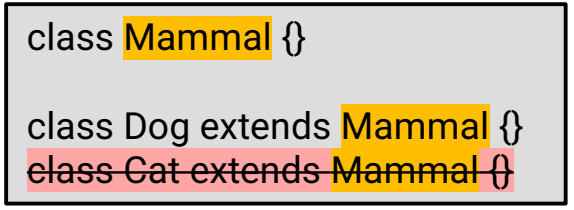

(Closest) Common Ancestor
class Animal {}
class Dog extends Animal {}
class Cat extends Animal {}
Change line 1,3,4
Change line 1,3,4
Add line 5
Remove line 4
class Pet {}
class Dog extends Pet {}
class Cat extends Pet {}
class Mammal {}
class Dog extends Mammal {}
class Cat extends Mammal {}
class Animal {}
class Dog extends Animal {}
class Cat extends Animal {}
class Giraffe extends Animal {}
class Animal {}
class Dog extends Animal {}
class Cat extends Animal {}
Conflict

Conflict terminology

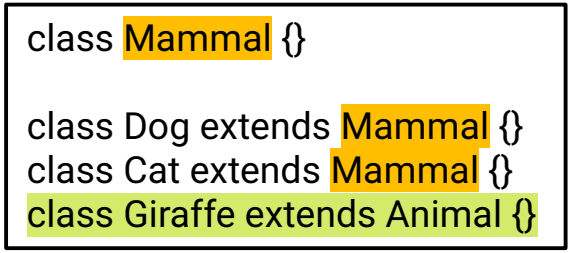
	class Animal {} class Dog extends Animal {} ~~class Cat extends Animal {}~~ class Giraffe extends Animal {}	class Pet {} class Dog extends Pet {} class Cat extends Pet {}	class Mammal {} class Dog extends Mammal {} ~~class Cat extends Mammal {}~~	? example will be given later...
conflict detected?	negative	positive	positive	negative
detection correct?	true	true	false	false
so let's call it a ...	***true negative***	***true positive***	***false positive***	***false negative***

Limitations of text-based versioning

- Problems encountered:



(a)



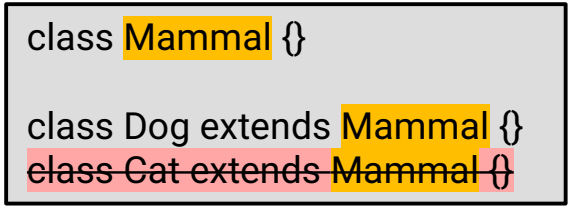
(b)

(a) “false positive” conflicts (common in git)

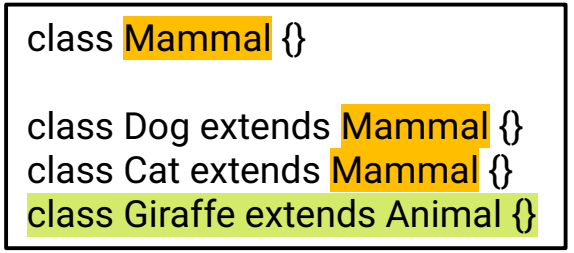
(b) undesired merge outcome

Limitations of text-based versioning

- Problems encountered:



(a)



(b)

(a) “false positive” conflicts (common in git)

(b) undesired merge outcome

- Going beyond git
  1) Versioning the abstract syntax (instead of concrete syntax)
  2) Operation-based versioning (instead of snapshot-based)

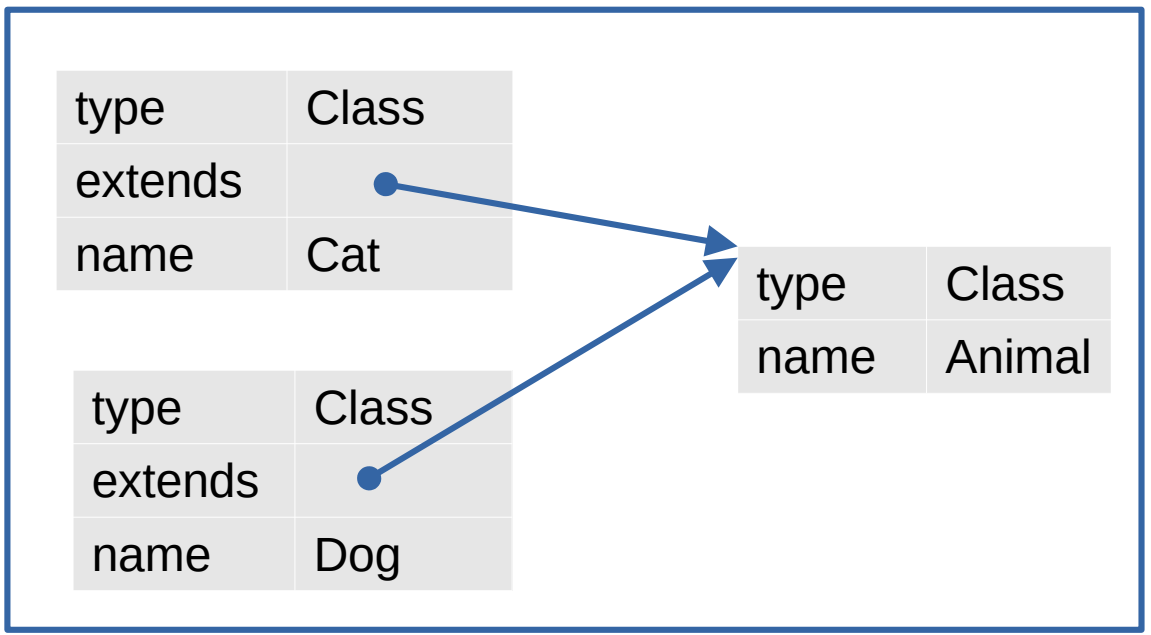
Beyond git: 1) Versioning the abstract syntax

concrete syntax

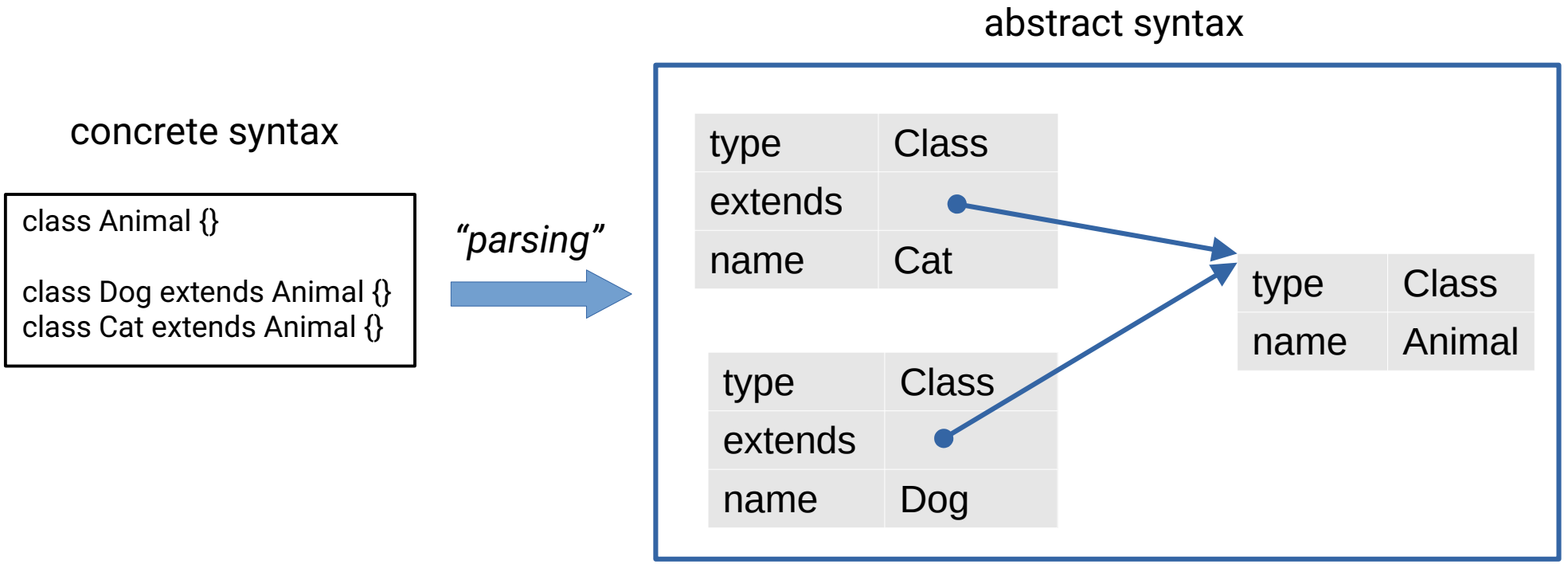
```
class Animal {}

class Dog extends Animal {}
class Cat extends Animal {}
```

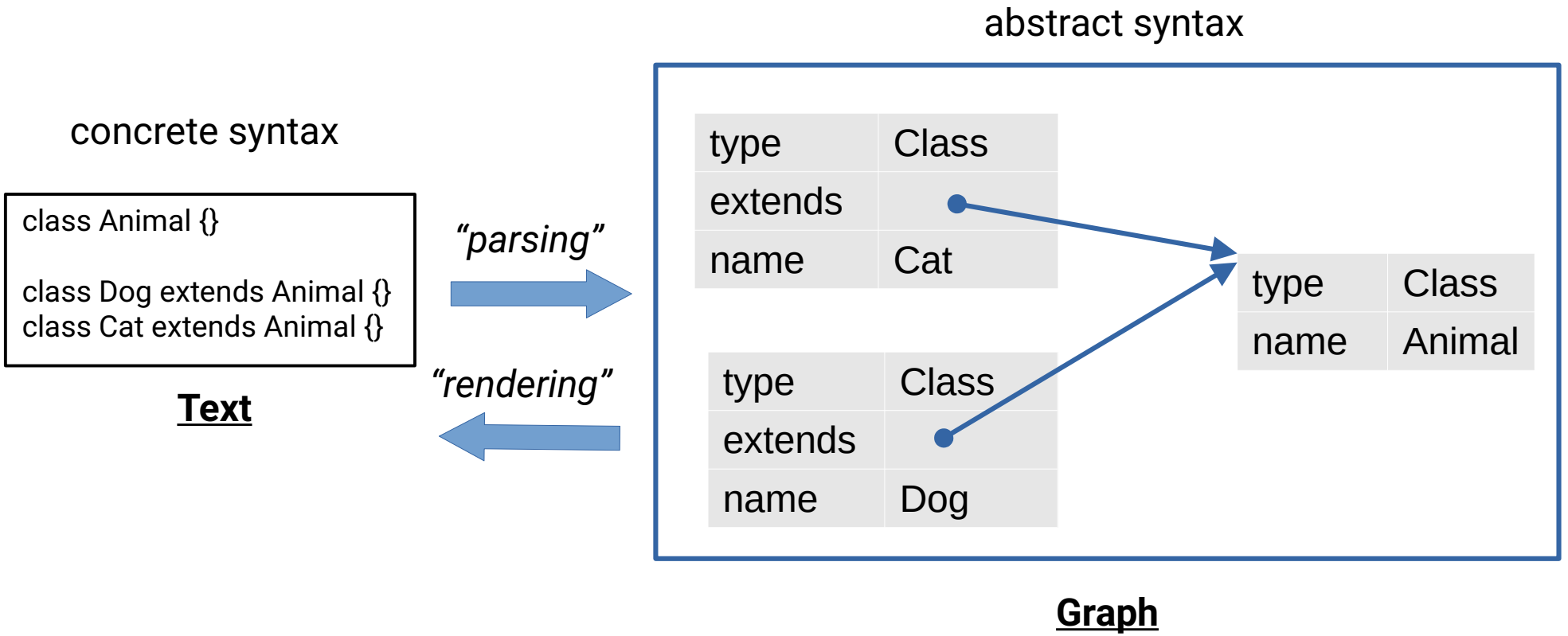
abstract syntax

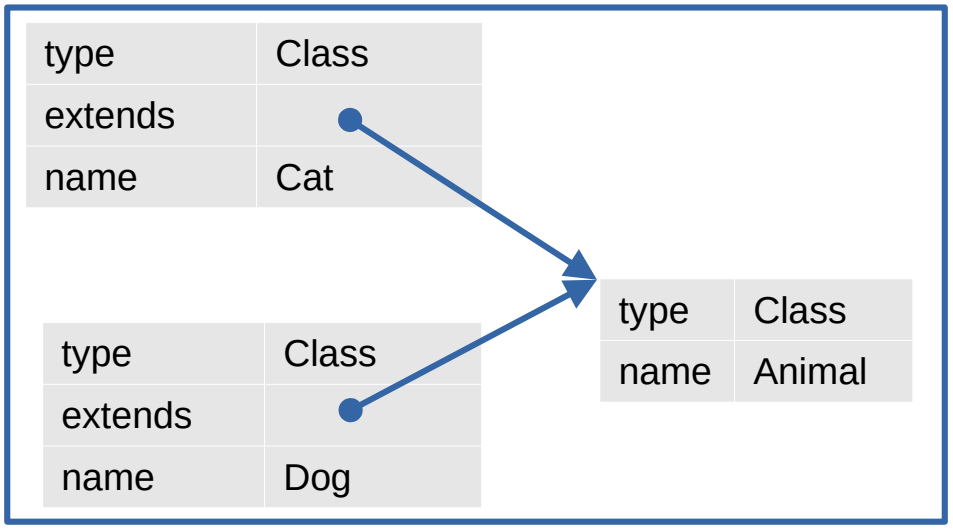


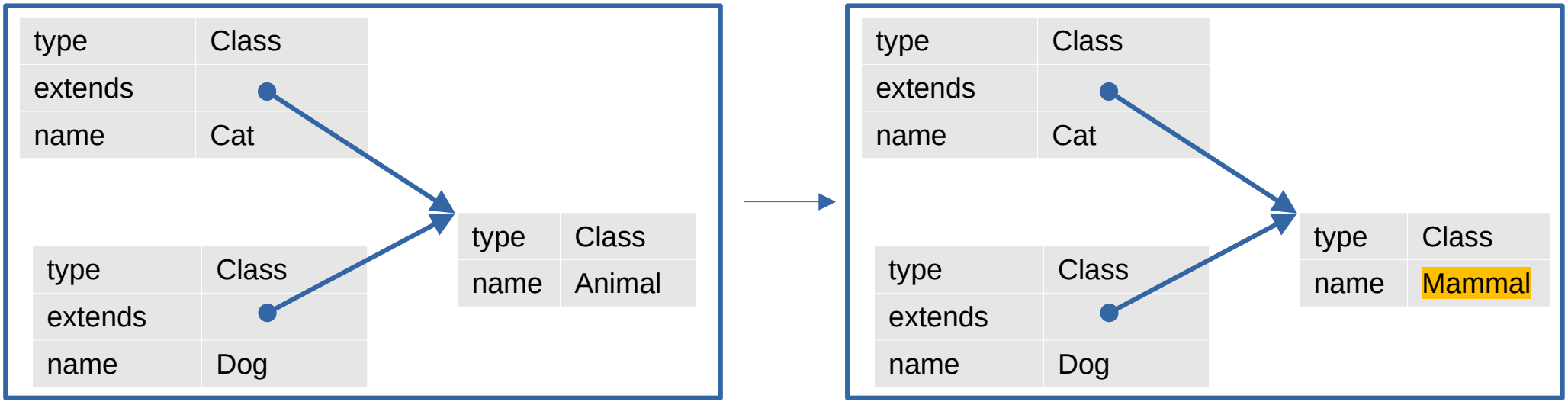
Beyond git: 1) Versioning the abstract syntax

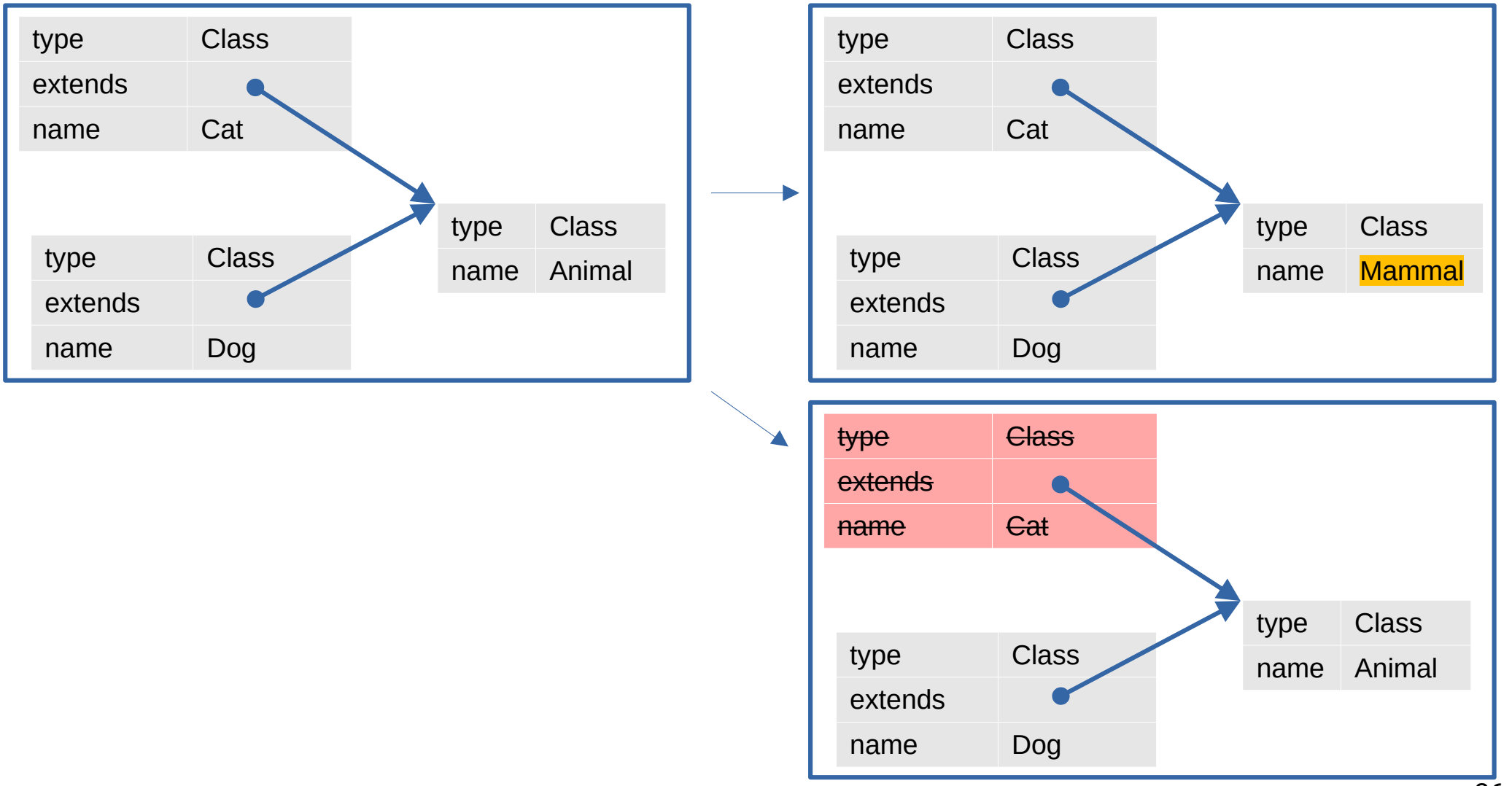


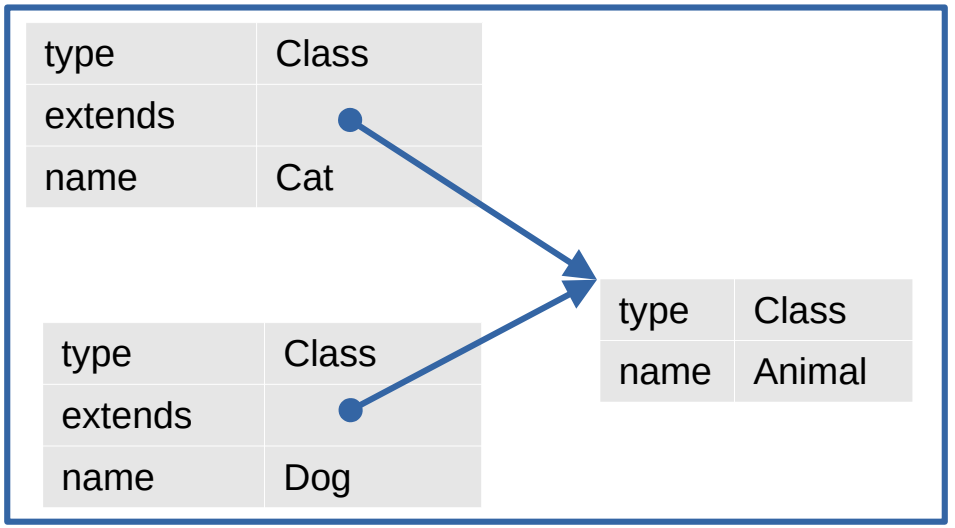
Beyond git: 1) Versioning the abstract syntax

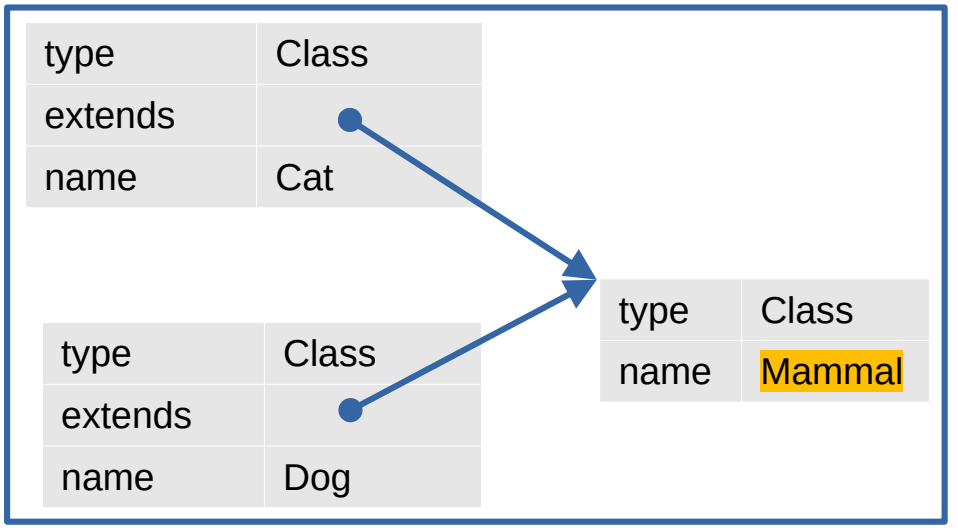



type
Class
extends
name
Cat
type
Class
name
Animal
type
Class
extends
name
Dog

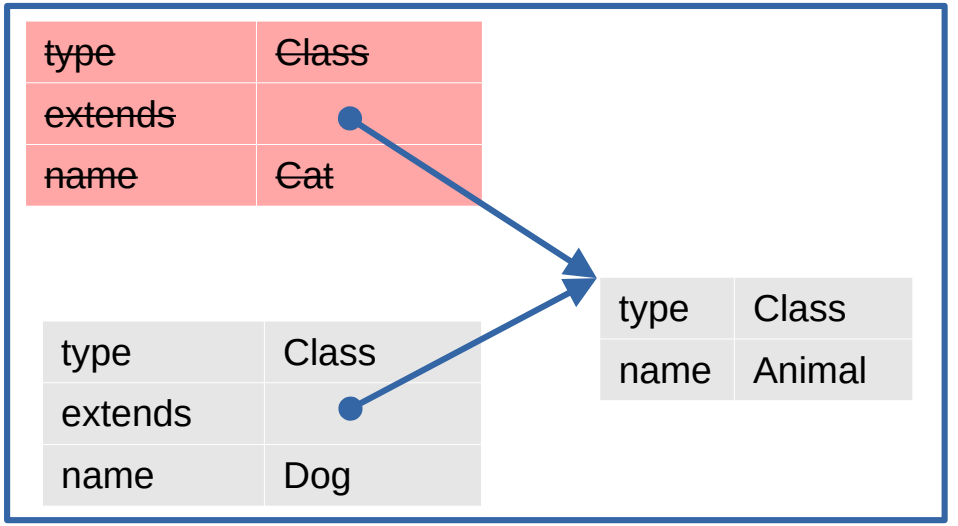

type
Class
extends
name
Cat
type
Class
extends
name
Dog
type
Class
name
Animal
type
Class
extends
name
Cat
type
Class
extends
name
Dog
type
Class
name
Mammal


type
Class
extends
name
Cat
type
Class
name
Animal
type
Class
extends
name
Dog
type
Class
extends
name
Cat
type
Class
name
Mammal
type
Class
extends
name
Dog
type
Class
extends
name
Cat
type
Class
name
Animal
type
Class
extends
name
Dog





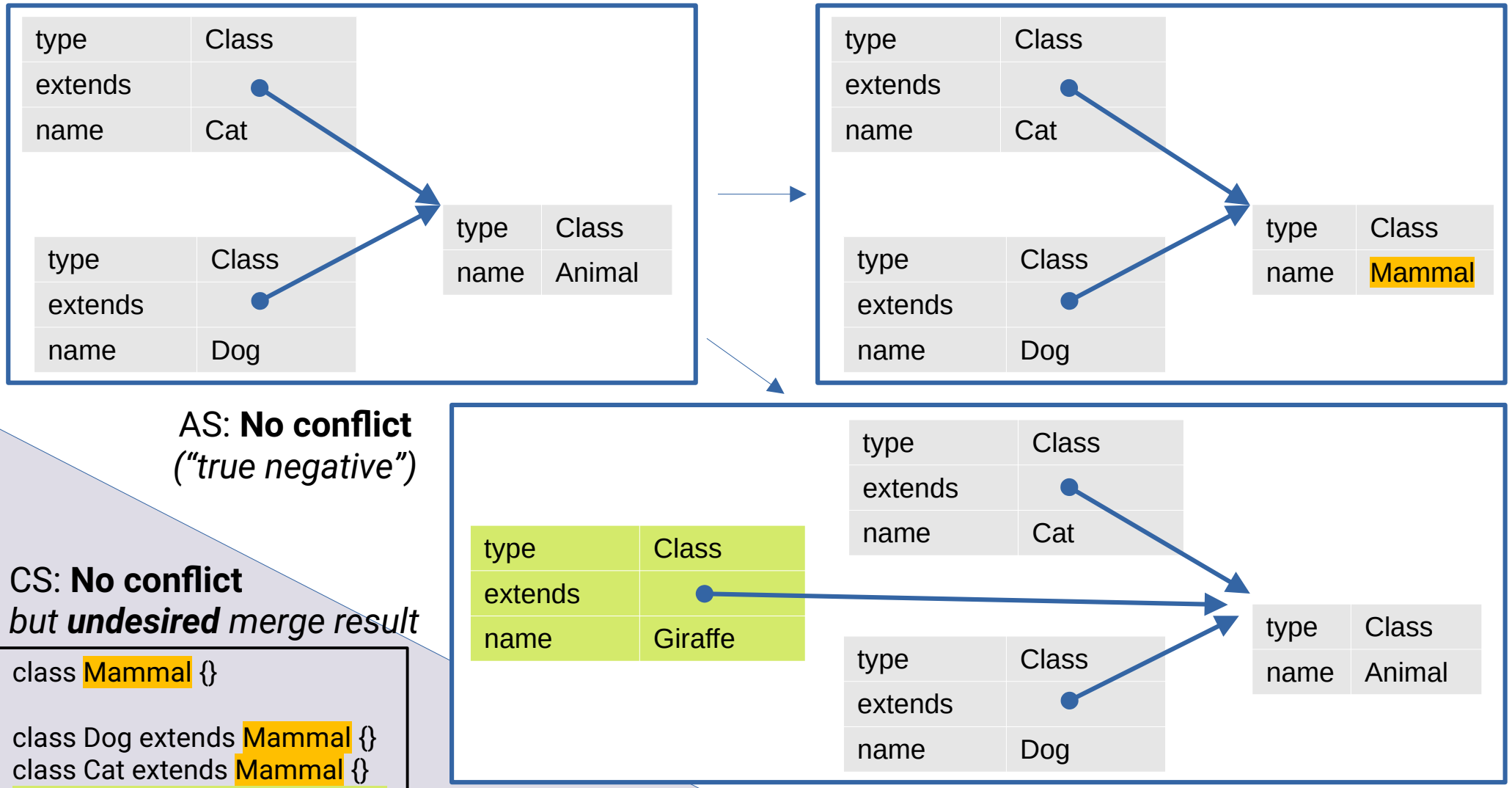
AS: **No conflict**
("true negative")



CS: **Conflict**
("false positive")

class Mammal {}

class Dog extends Mammal {}
~~class Cat extends Mammal {}~~


type Class
extends
name Cat
type Class
extends
name Dog
type Class
name Animal
type Class
extends
name Cat
type Class
extends
name Dog
type Class
name Mammal
AS: No conflict
("true negative")
type Class
extends
name Giraffe
type Class
extends
name Cat
type Class
extends
name Dog
type Class
name Animal
CS: No conflict
but undesired merge result
class Mammal {}
class Dog extends Mammal {}
class Cat extends Mammal {}
class Giraffe extends Animal {}

Beyond git: 1) Versioning the abstract syntax

- Benefits
  - Fewer “false positive” conflicts
  - Abstract syntax closer to semantics of language
    => captures better the intention
    => better merge results

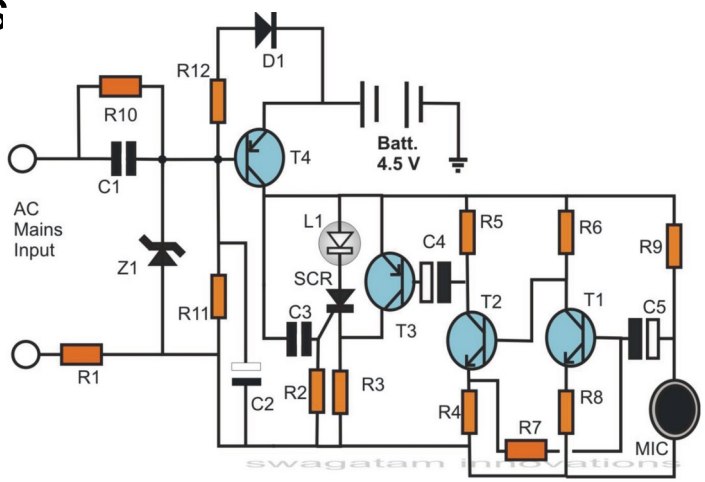
- However… (next slide)

Beware: only versioning the abstract syntax is undesirable

- Concrete syntax contains **more** information than abstract syntax
  - e.g., textual syntax: order of declarations
  - e.g., visual syntax: layout of elements

```
class Animal {}

class Dog extends Animal {}
class Cat extends Animal {}
```

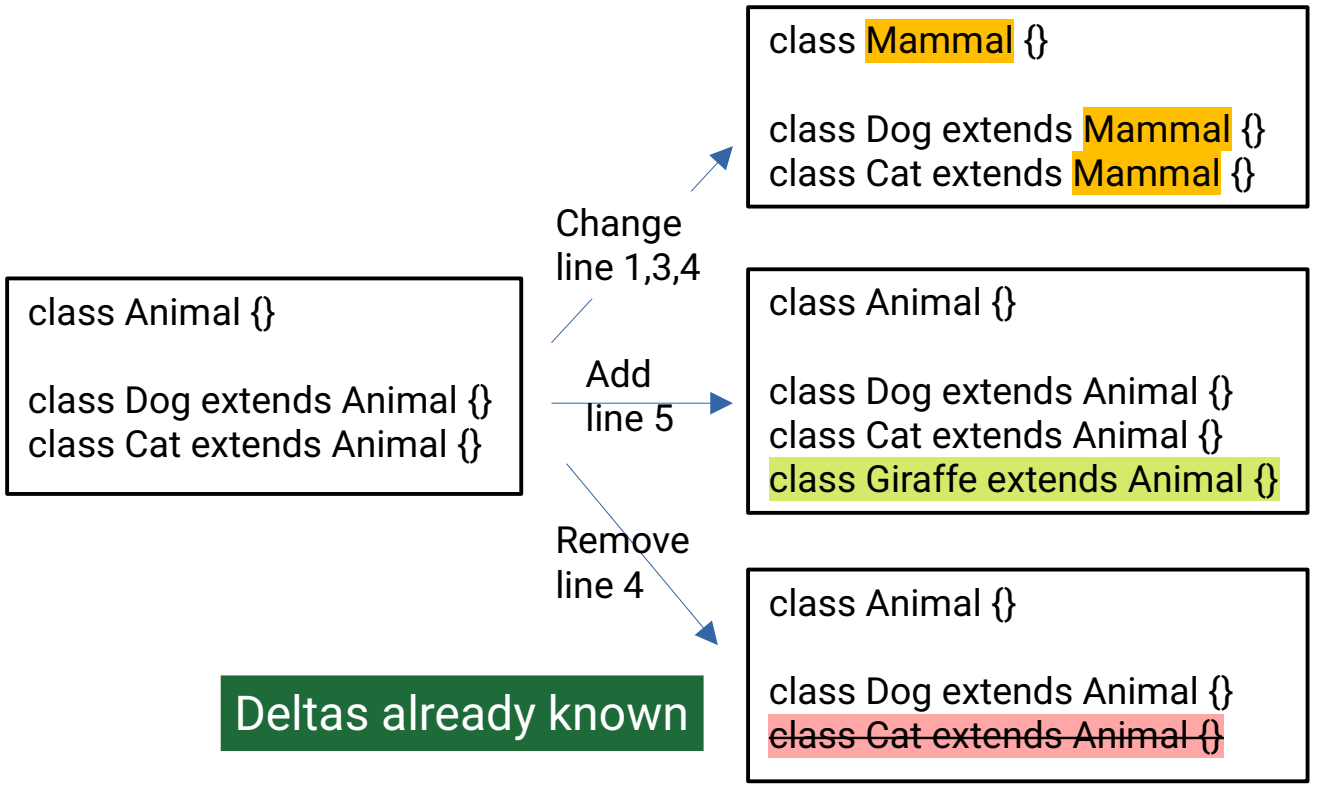


- This information is **useful** for **humans**
- Losing this information leads to ***layout discontinuity***

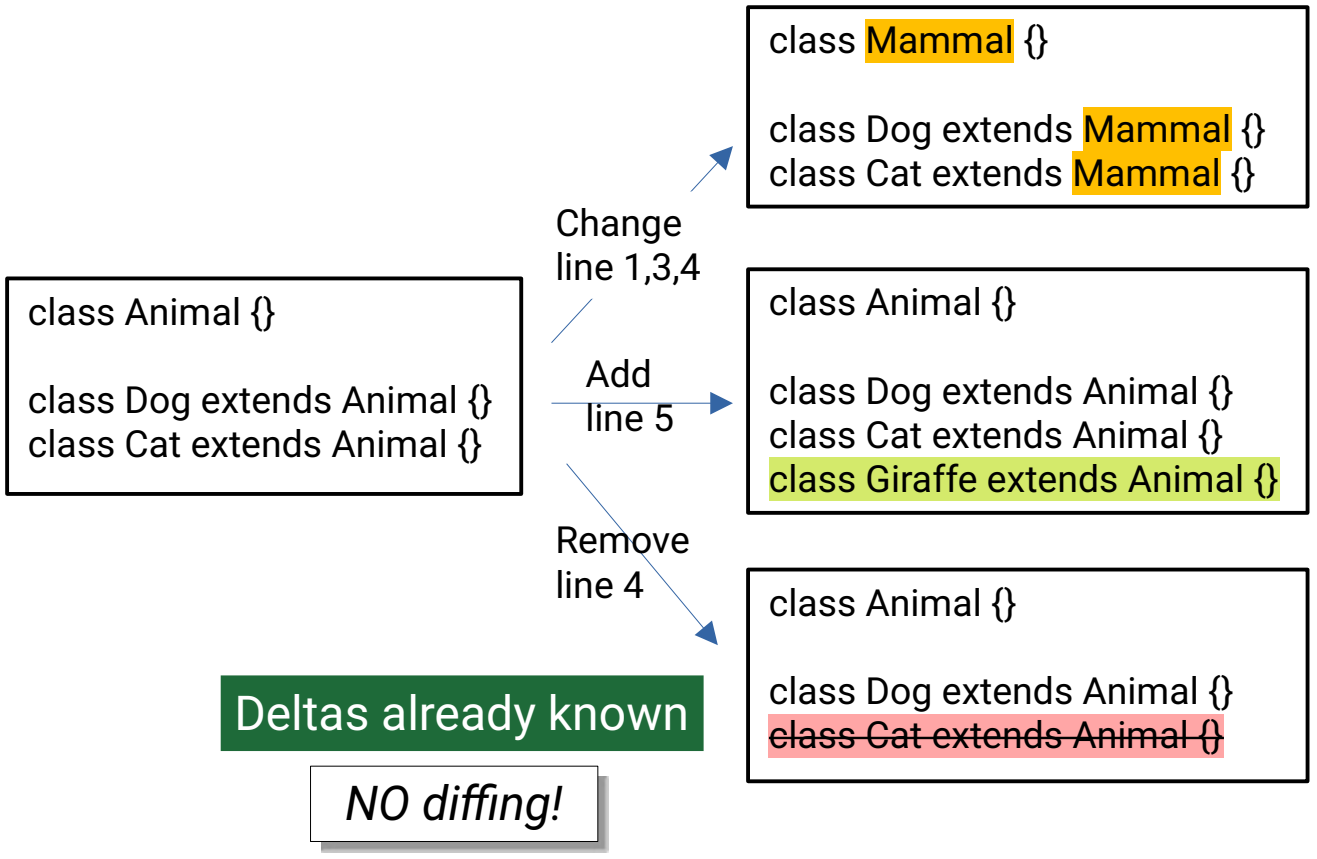
Beyond git: 2) Operation-based versioning

- Persist deltas immediately after they happen
  - no diffing
  - no need to record snapshots (can be restored from the deltas)
  - Version == sequence of deltas
- Can compute conflicts between deltas in advance (before merging)
  - can persist conflict relations

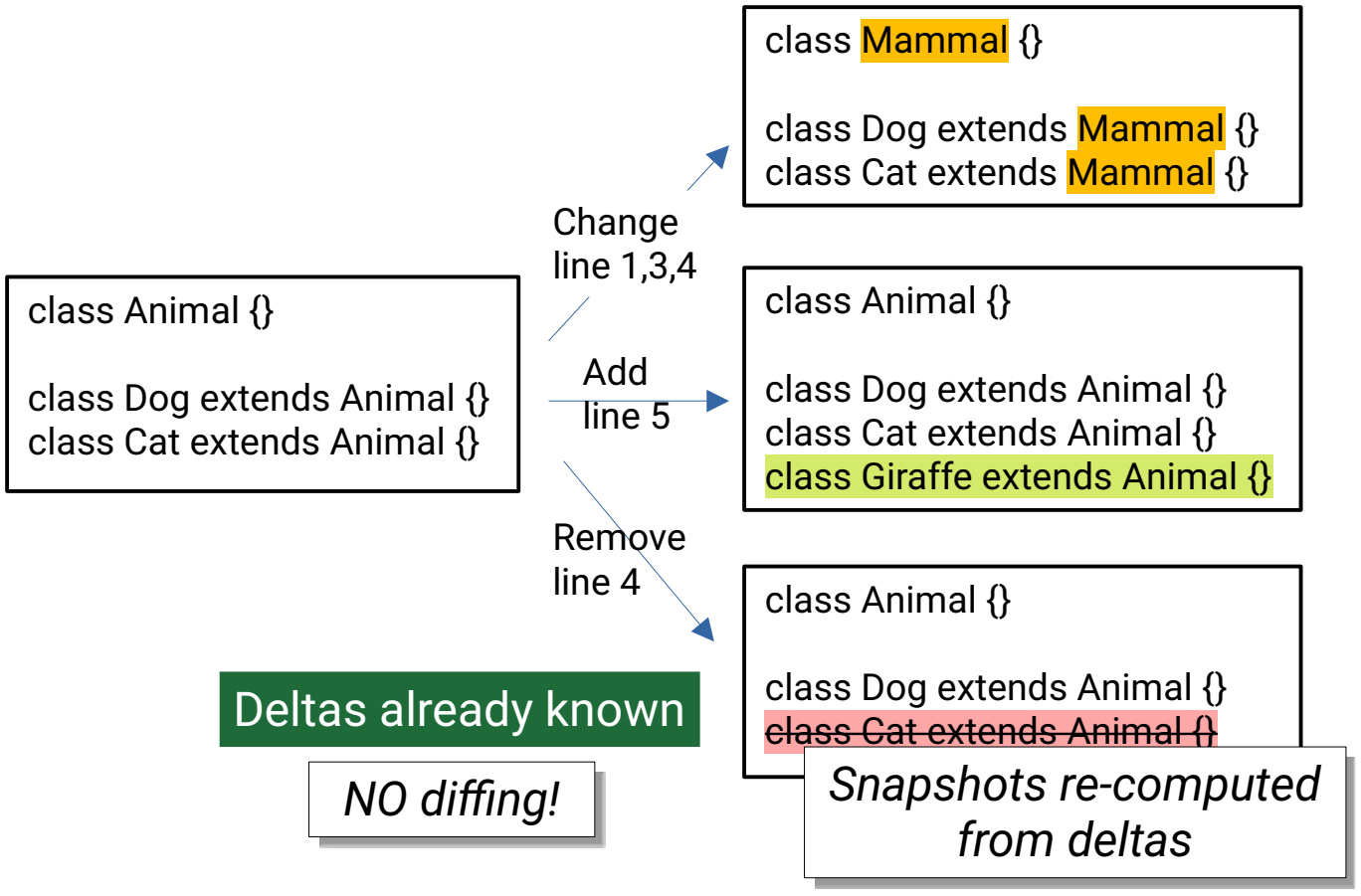
Beyond git: 2) Operation-based versioning



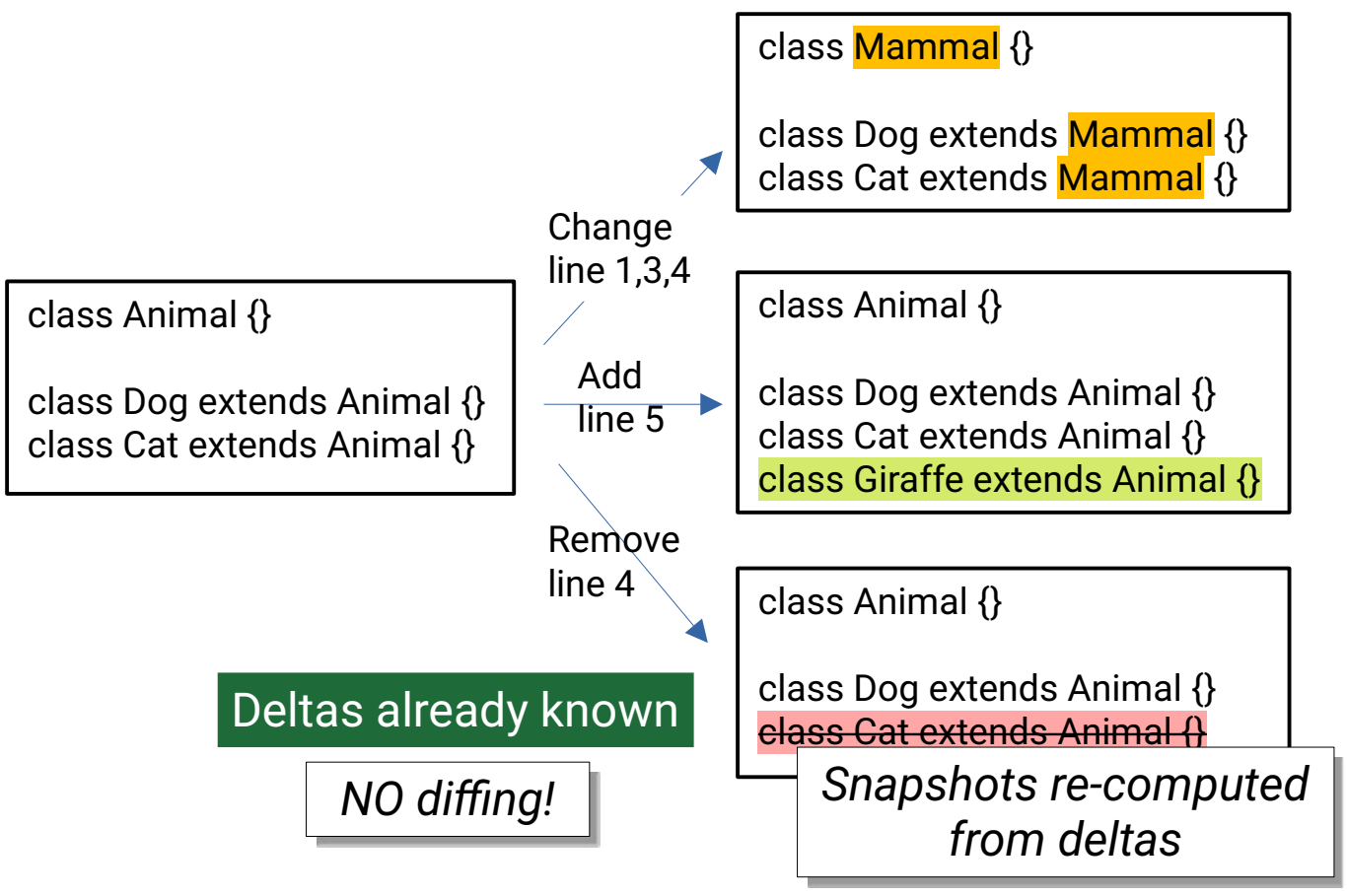
Beyond git: 2) Operation-based versioning

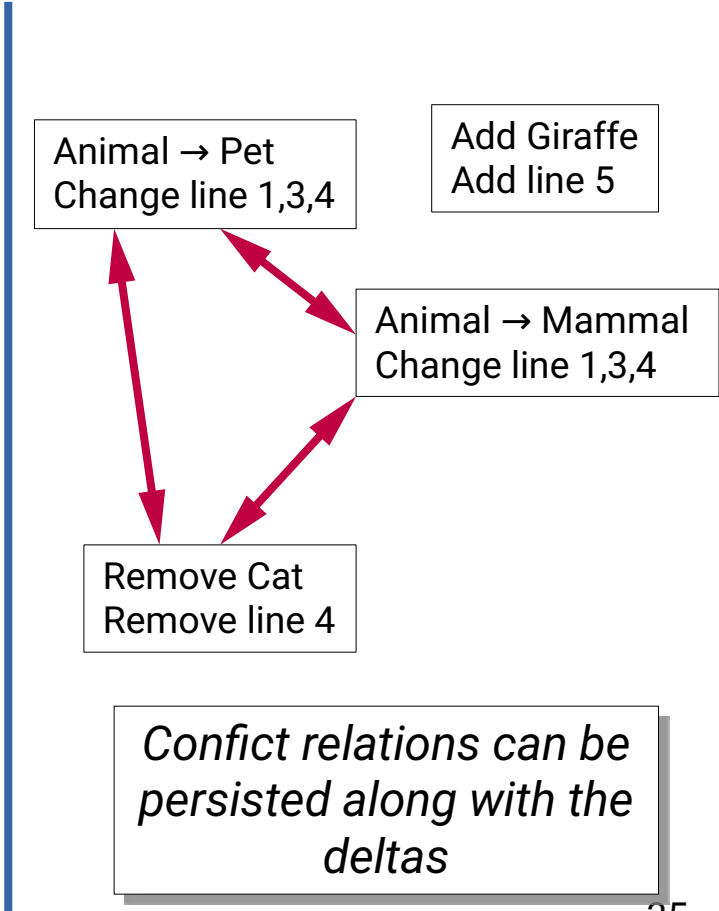


Beyond git: 2) Operation-based versioning

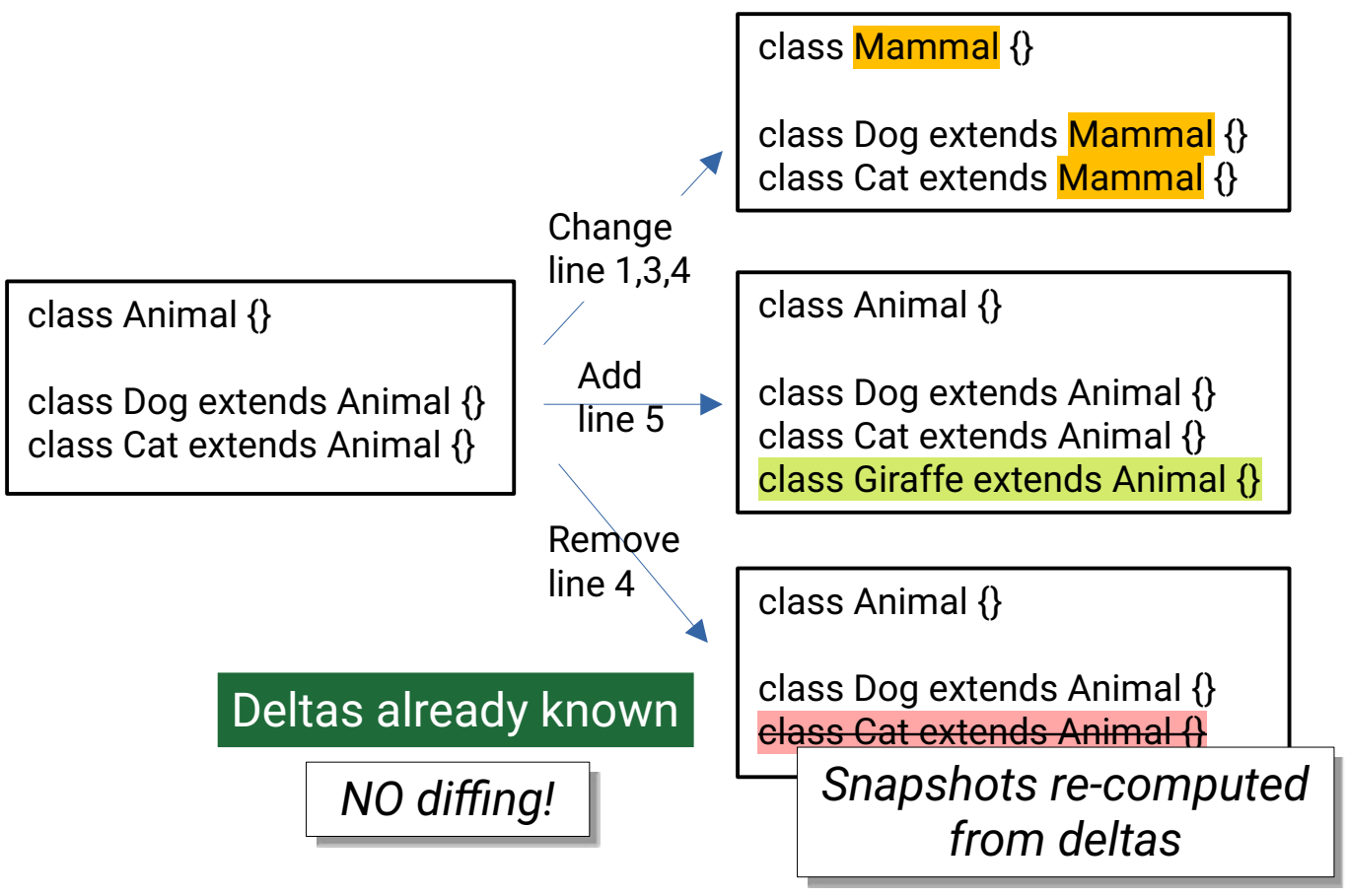


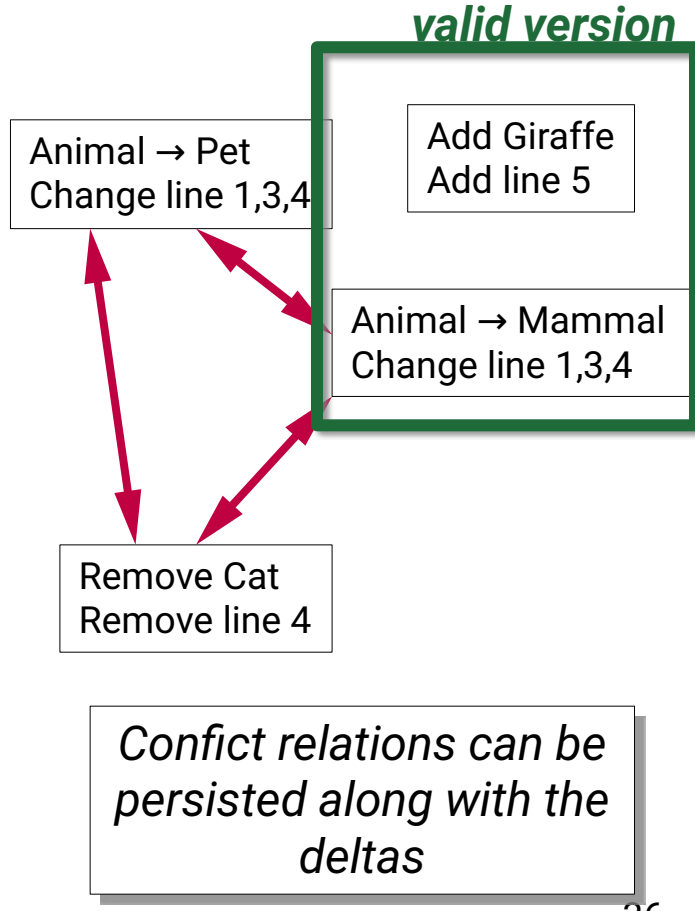
Beyond git: 2) Operation-based versioning



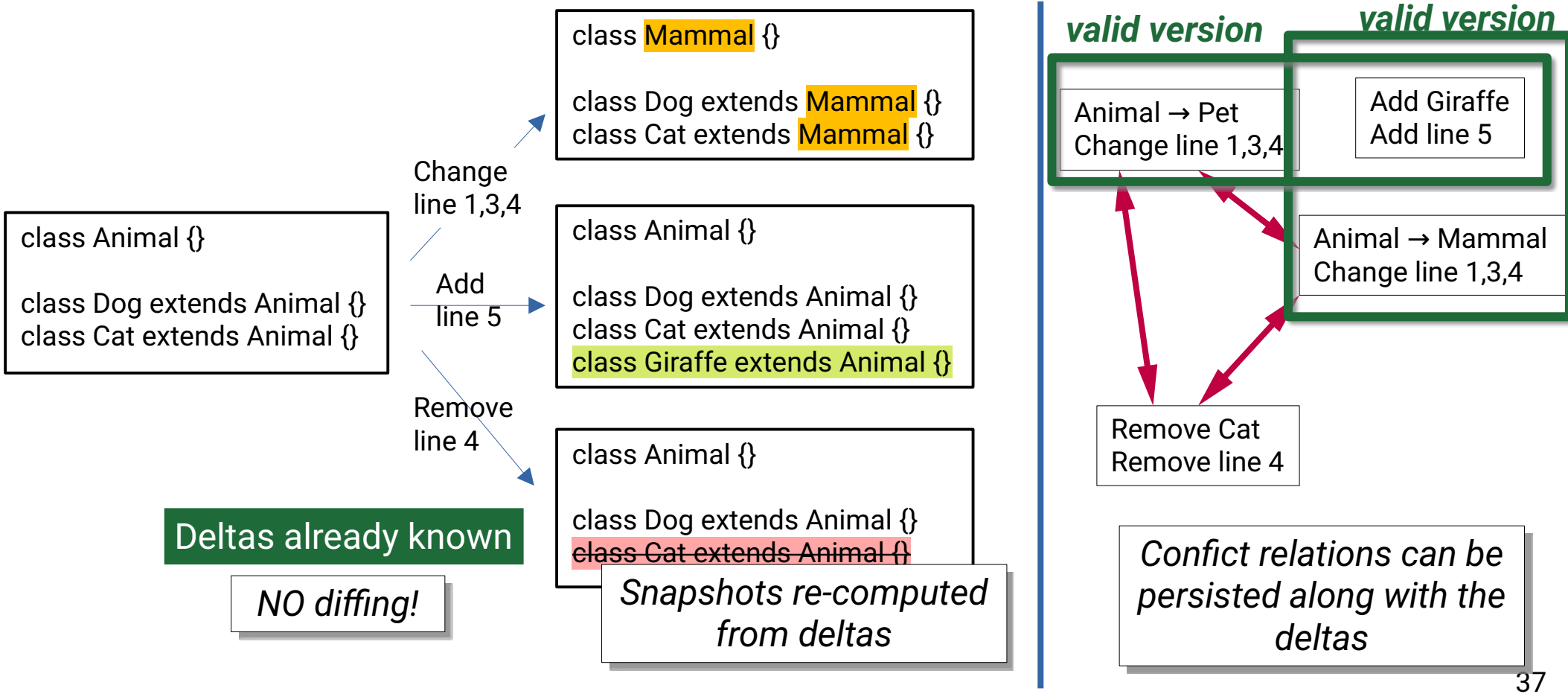


Beyond git: 2) Operation-based versioning

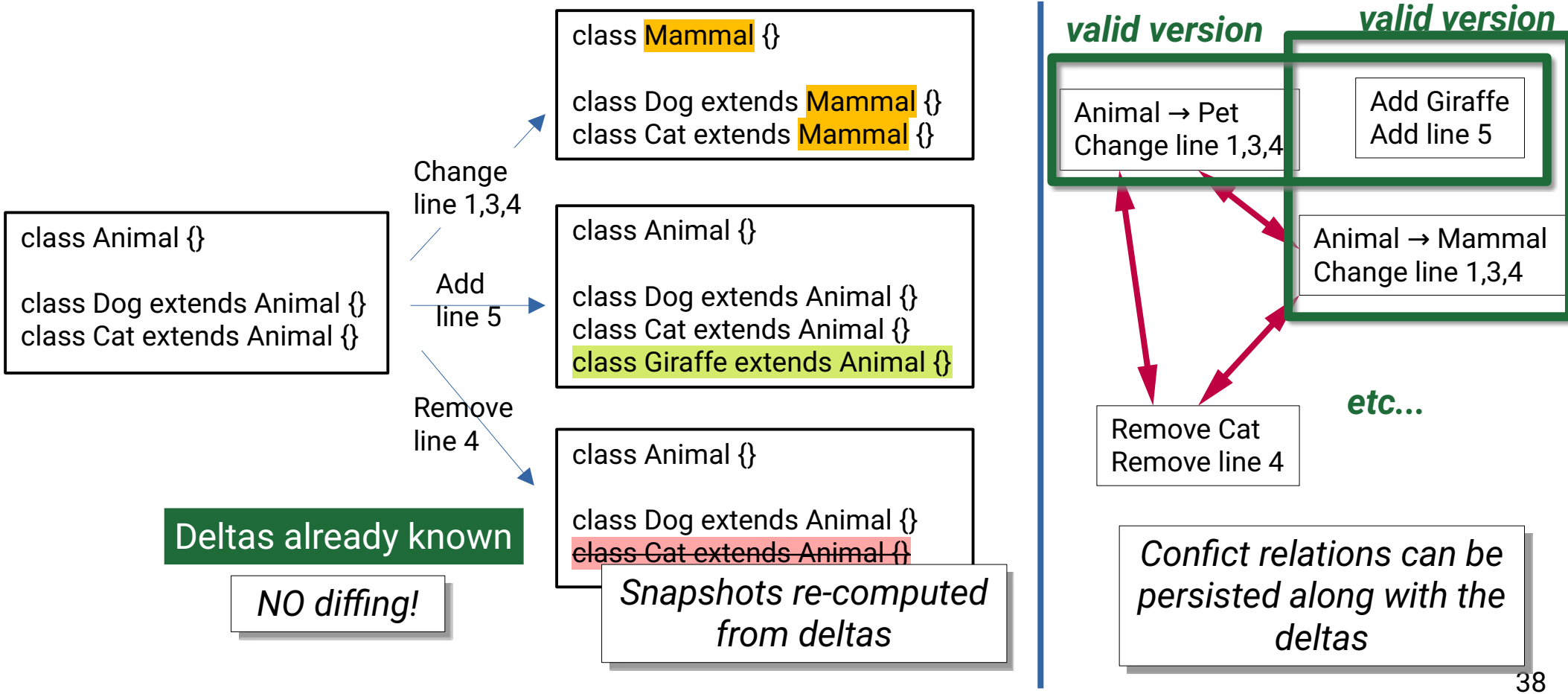




Beyond git: 2) Operation-based versioning



Beyond git: 2) Operation-based versioning



Beyond git: 2) Operation-based versioning

- Benefits
  - **More efficient** (does the work up front)
    - no diffing
    - no need to store snapshots (containing duplicate information)
  - Can persist **additional information** about deltas
    - conflicts
    - dependencies (explained later...)
  - Fine-grained edit history (every edit operation is recorded)
    - can also become a downside: drowning in information…
  - Need deltas **not only for merging**, but also:
    - “live” collaboration (e.g., Google Docs)
      - send deltas over network, rather than state snapshots
    - incremental transformations (e.g., parsing, code generation, ...)

Beyond git: Related work

- Darcs (https://darcs.net/)
  - alternative to git (textual versioning)
  - persists deltas and their *dependencies* (explained later…)
    (performs eager diffing)
  - written in Haskell

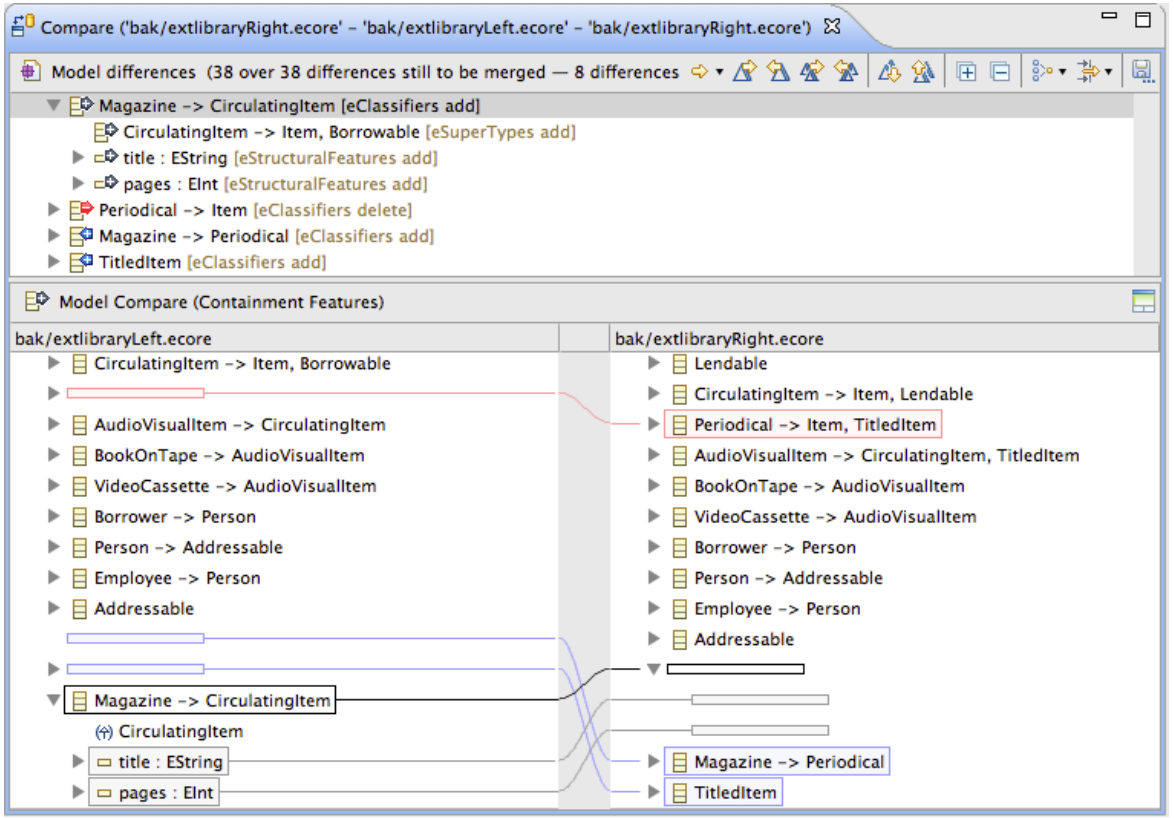
- Pijul (https://pijul.org/)
  - inspired by and similar to Darcs, but claims better performance
  - “theoretically sound”
  - written in Rust

Background: Versioning in model-driven engineering

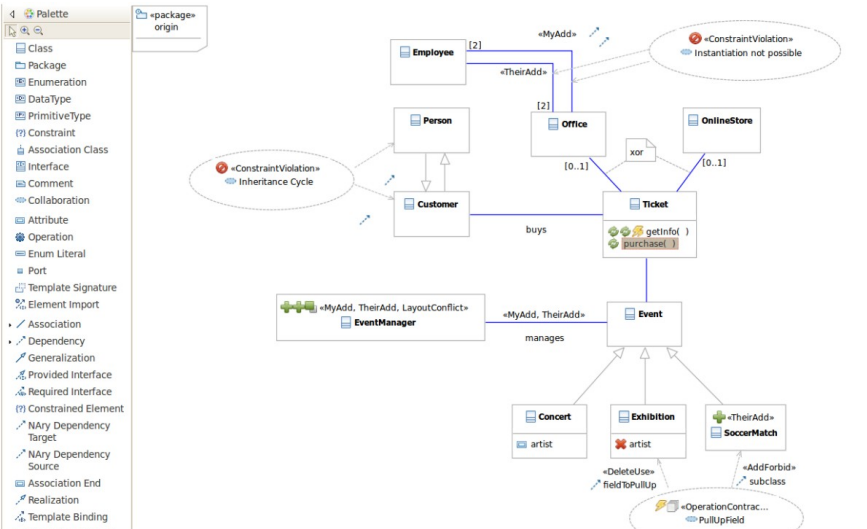
- **Text-based** versioning (e.g., git, SVN) is a poor fit for MDE
  - for models, can only take snapshots of *serializations* (e.g., XML)
  - often have “false positive” conflicts

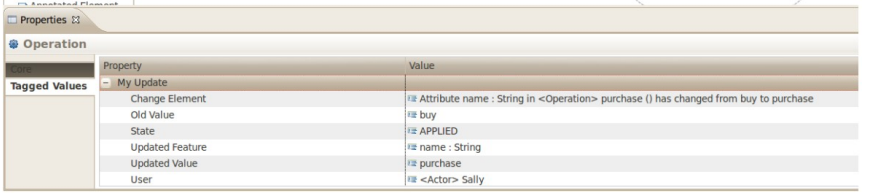
- **Model-specific** versioning solutions work primarily at the level of **abstract syntax**
  - a 1:1 mapping between concrete and abstract syntax elements is assumed (concrete syntax == abstract syntax + icons)

    => very little “concrete syntax freedom”



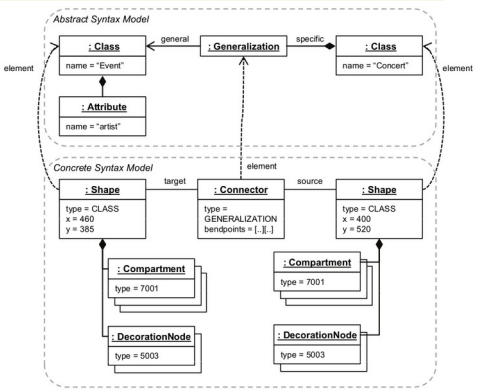
EMF Compare[1]





AMOR
Adaptable Model Versioning

2



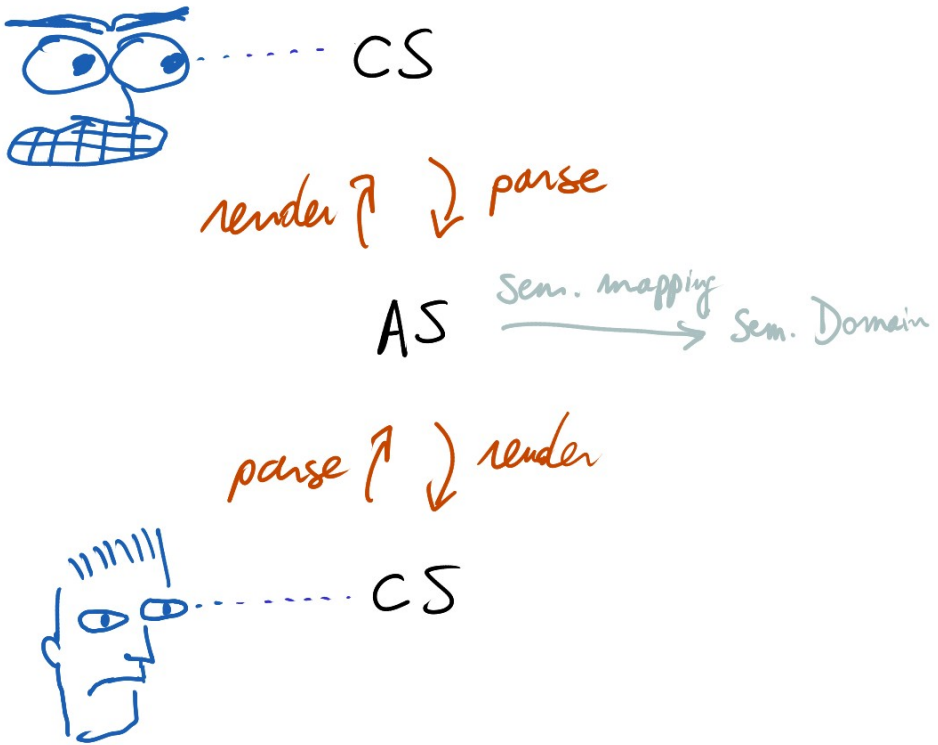
[1] https://www.eclipse.org/emf/compare/

[2] Petra Brosch, Martina Seidl, Manuel Wimmer, Gerti Kappel: **Conflict Visualization for Evolving UML Models**. J. Object Technol. 11(3): 2: 1-30 (2012)

Background: Blended modeling[1]

= Ability to *edit* a model (= abstract syntax)
through *multiple* concrete syntaxes

- Usability++: User can choose the **representation** that is **most efficient** for the current task
  - e.g., understanding connections between elements
    -> observe visual layout
  - e.g., renaming a variable
    -> find all/replace in textual CS
- Technical challenge: **<u>incremental</u> bi-directional synchronization** between CSs and AS



[1] Federico Ciccozzi, Matthias Tichy, Hans Vangheluwe, Danny Weyns: **Blended Modelling - What, Why and How**. MoDELS (Companion) 2019: 425-430

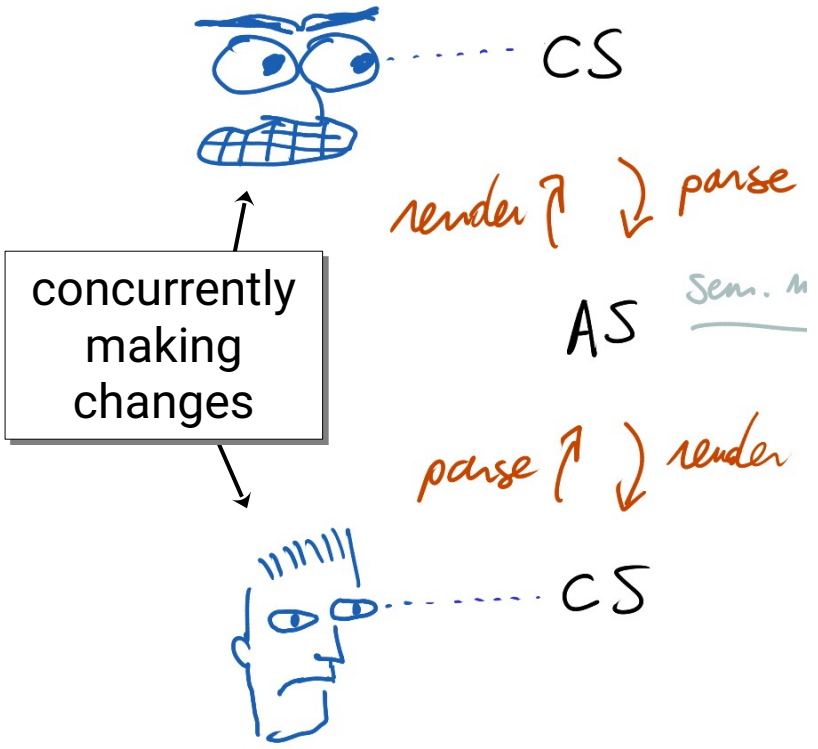
Research question

"What is needed to support collaborative blended modeling?"

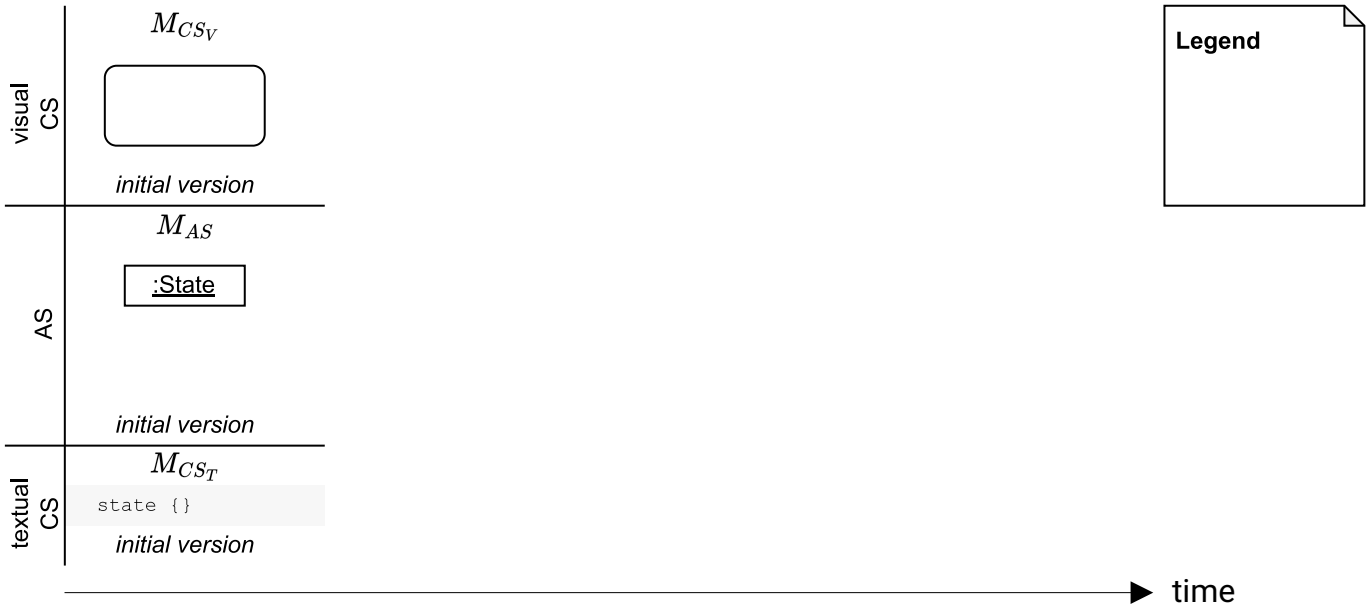
- branching
- detecting conflicts
- merging non-conflicting changes

At the level of:

- abstract syntax
- concrete syntaxes

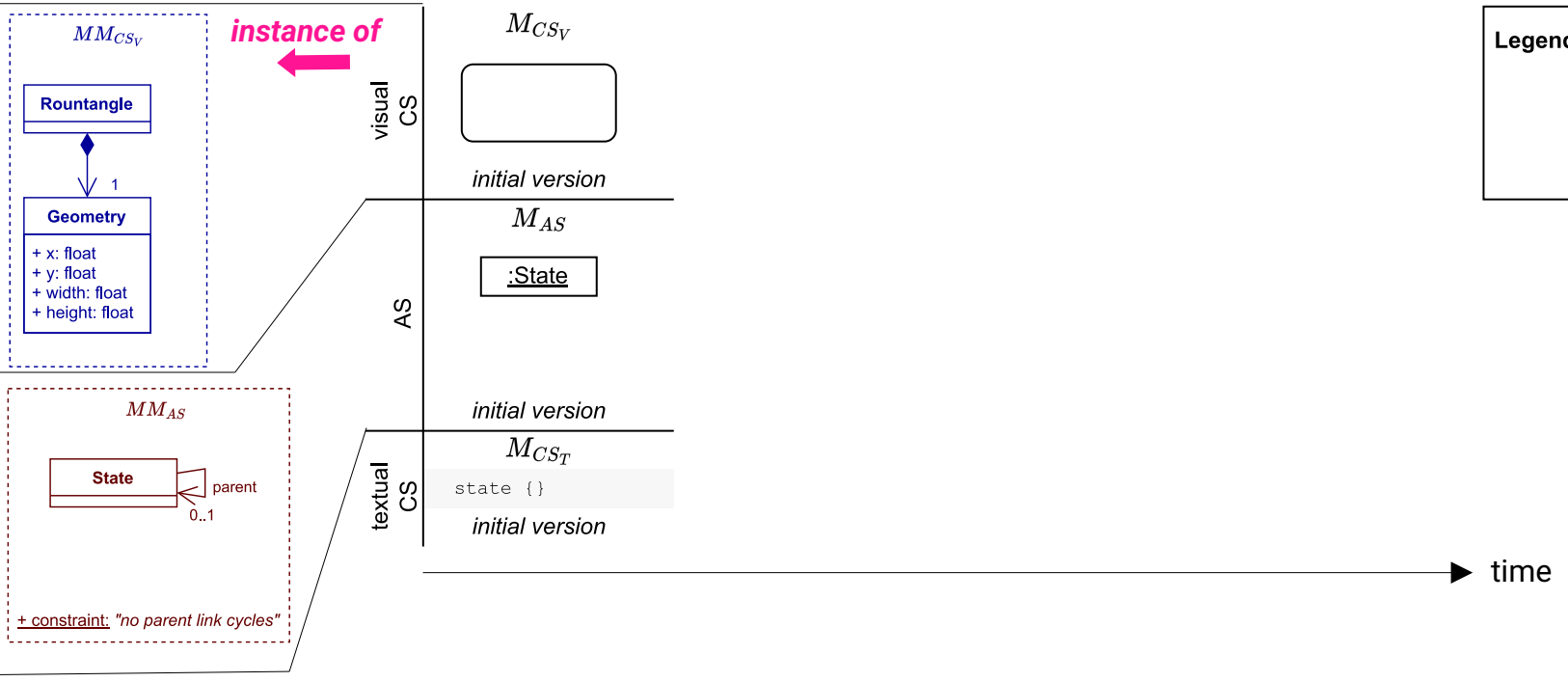


Running example: Concurrent edits × Blended modeling



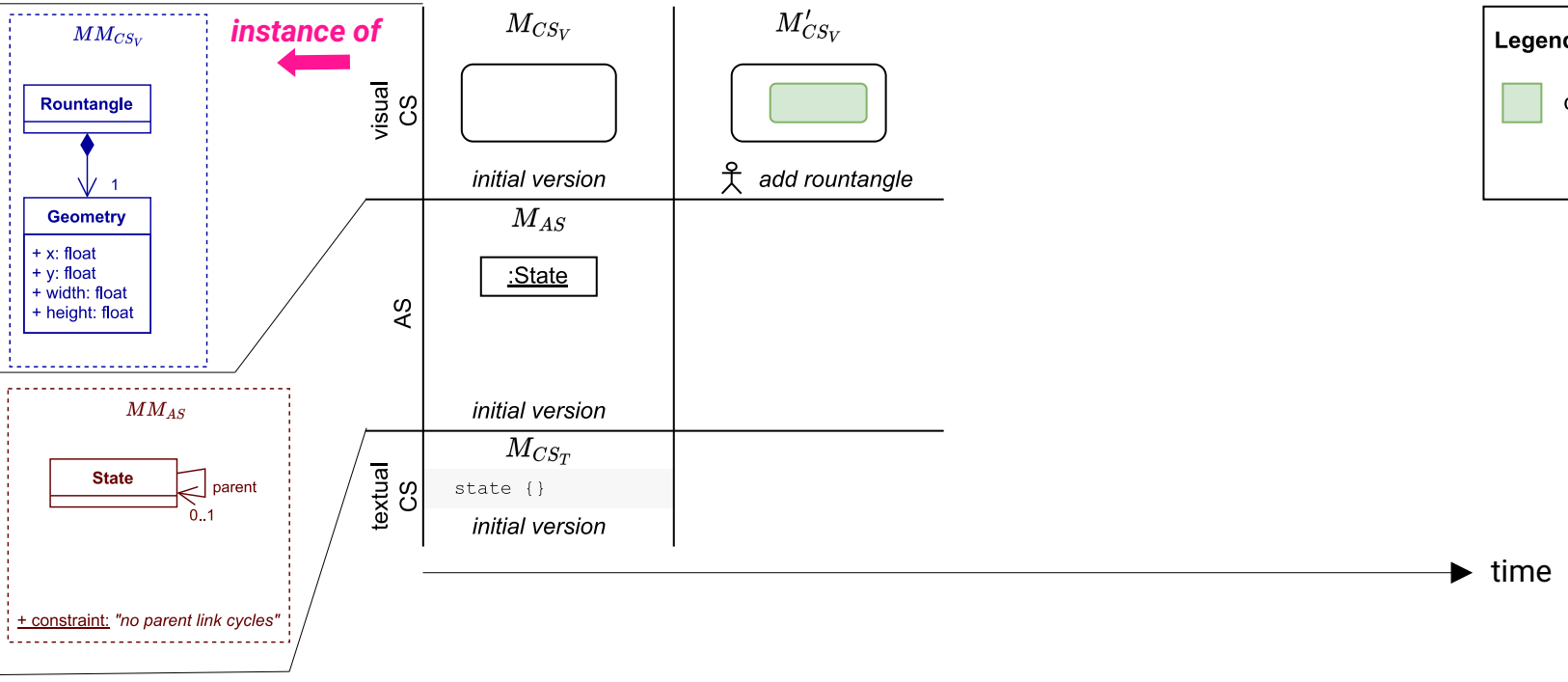
Term "rountangle" was introduced by: David Harel: **On Visual Formalisms**. Commun. ACM 31(5): 514-530 (1988)

Running example: Concurrent edits × Blended modeling



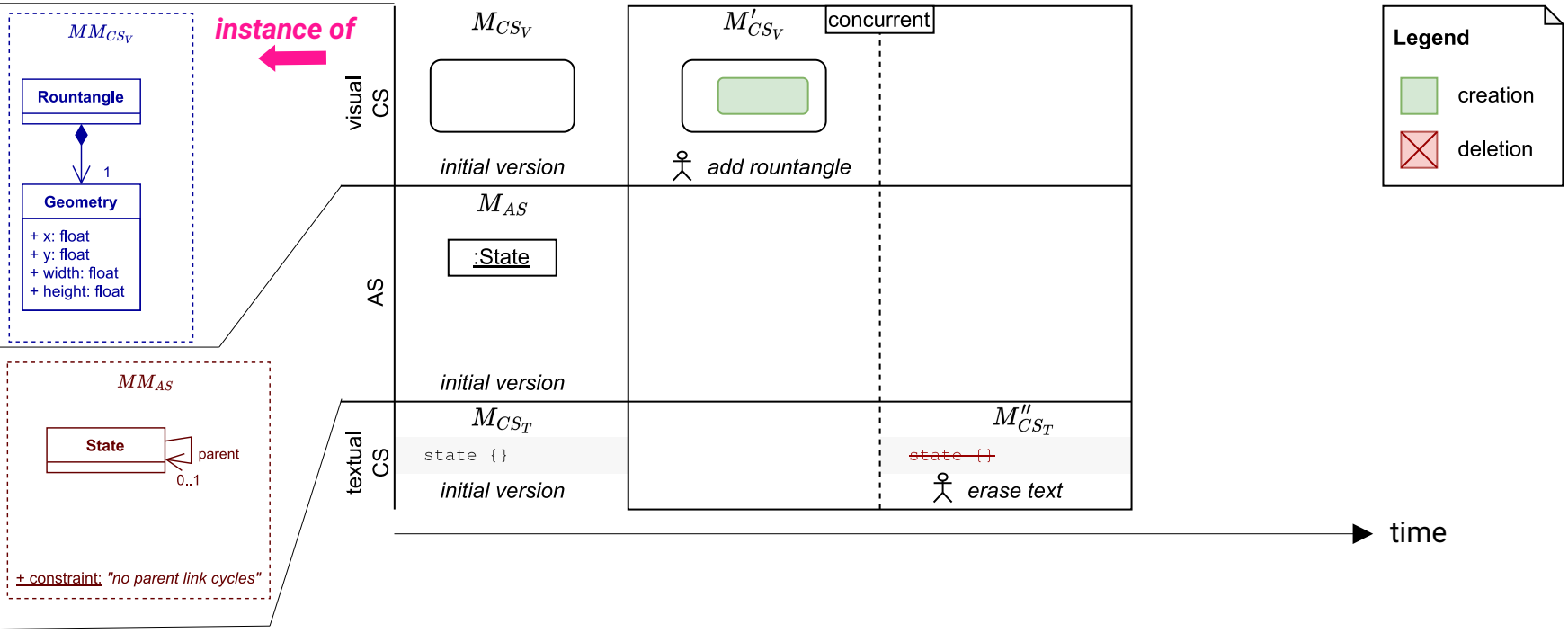
Term "rountangle" was introduced by: David Harel: **On Visual Formalisms**. Commun. ACM 31(5): 514-530 (1988)

Running example: Concurrent edits × Blended modeling



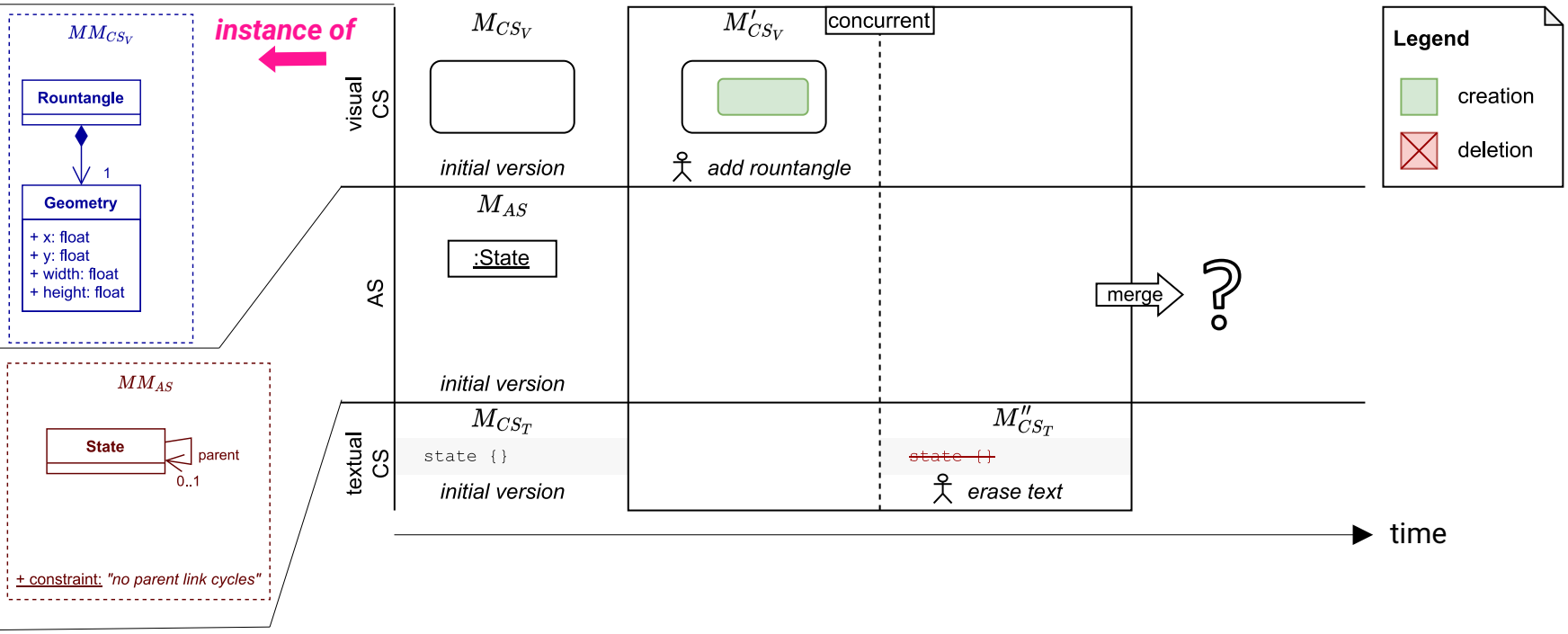
Term "rountangle" was introduced by: David Harel: **On Visual Formalisms**. Commun. ACM 31(5): 514-530 (1988)

Running example: Concurrent edits × Blended modeling



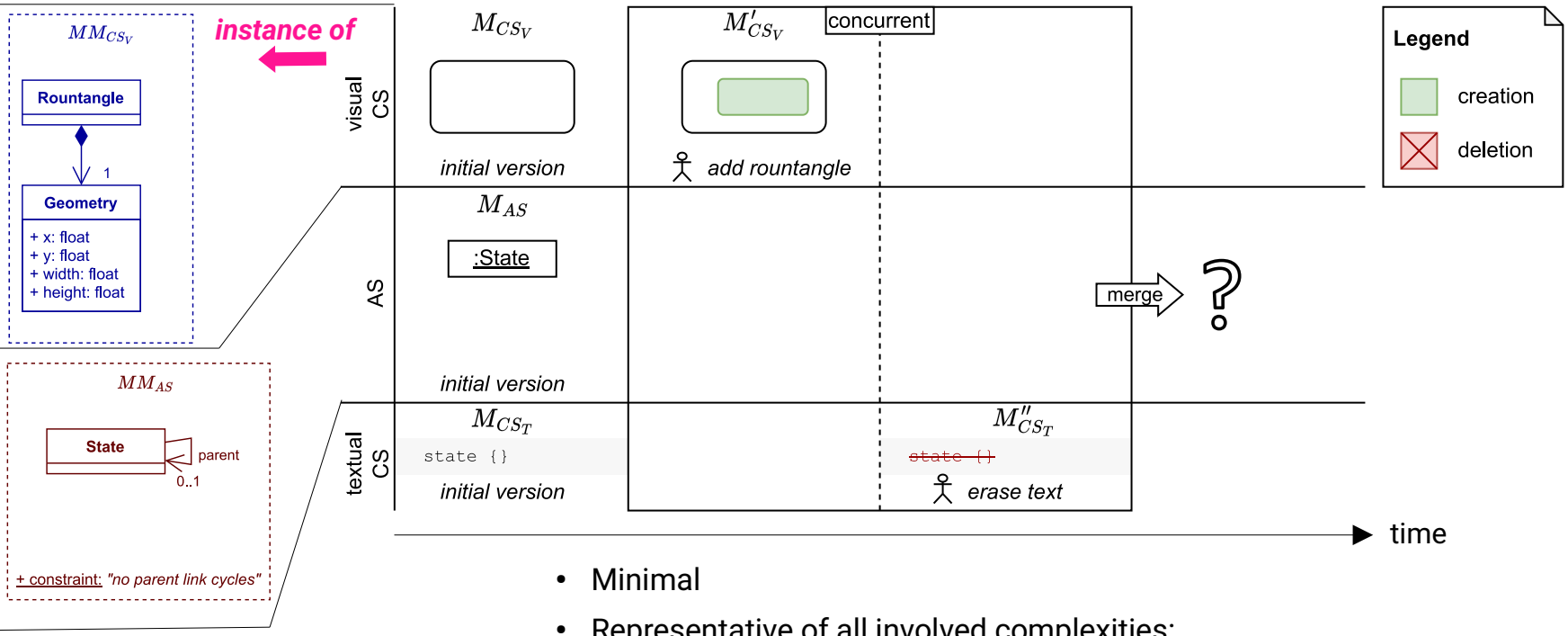
Term "rountangle" was introduced by: David Harel: **On Visual Formalisms**. Commun. ACM 31(5): 514-530 (1988)

Running example: Concurrent edits × Blended modeling



Term "rountangle" was introduced by: David Harel: **On Visual Formalisms**. Commun. ACM 31(5): 514-530 (1988)

Running example: Concurrent edits × Blended modeling



- Minimal
- Representative of all involved complexities:
  - Concurrent edit operations
  - Bi-directional change propagation (CS <-> AS <-> CS)
  - Non-trivial CS <-> AS mappings

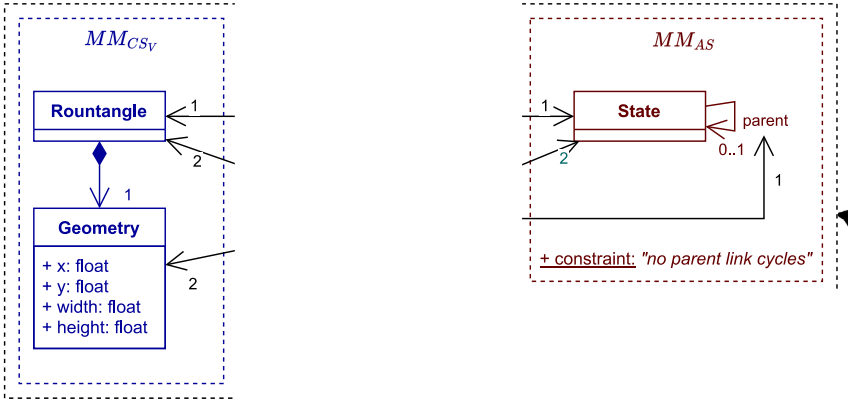
Term "rountangle" was introduced by: David Harel: **On Visual Formalisms**. Commun. ACM 31(5): 514-530 (1988)

Big picture of solution

- CS and AS are **both versioned**, and each conform to their **own metamodel** (inspired by [1])

  Running example:

  - visual CS metamodel: (vector) drawings (**reusable**!)
  - AS metamodel: Statecharts (simplified)

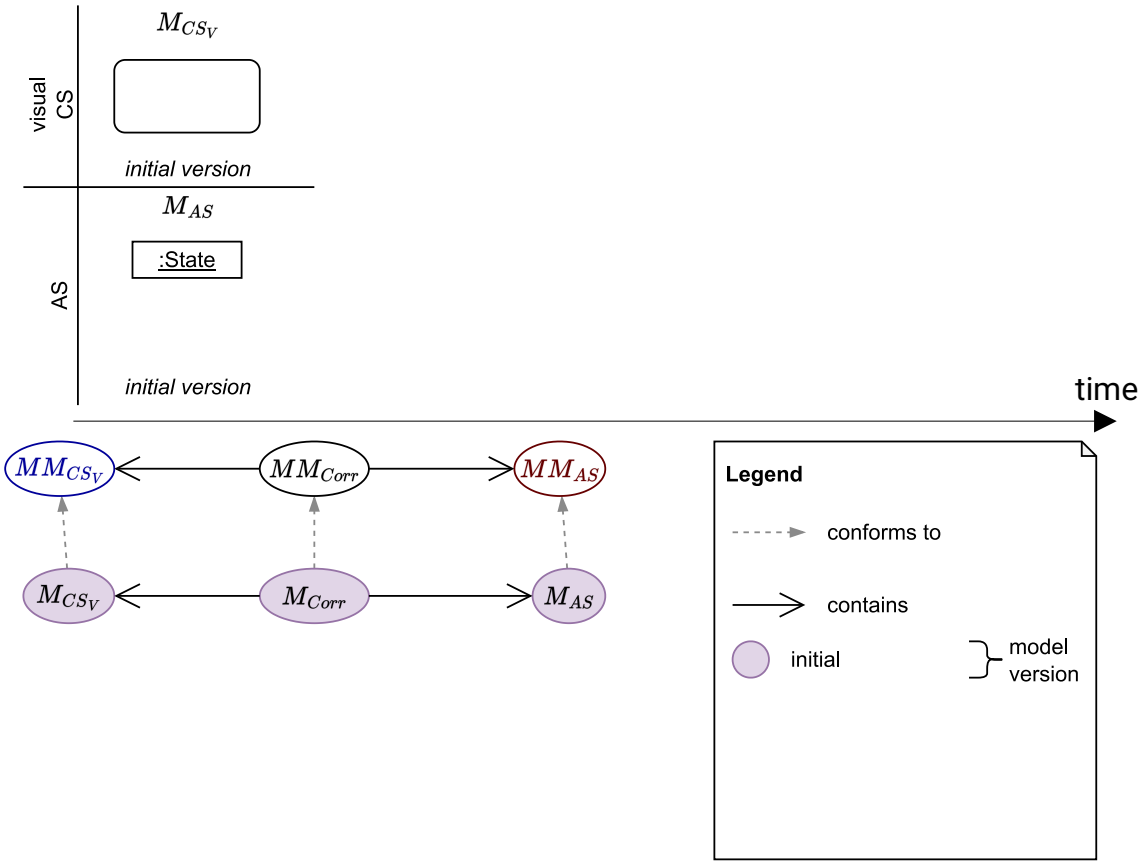


- Record correspondence links between CS and AS elements in **versioned correspondence model**
  - Idea taken from Triple Graph Grammars [2]
  - Correspondence model is **also versioned**, and also has its **own metamodel** (so-called “meta-triple”)

[1] Yentl Van Tendeloo, Hans Vangheluwe: **Unifying Model- and Screen Sharing**. WETICE 2018: 127-132
[2] Andy Schürr: **Specification of Graph Translators with Triple Graph Grammars**. WG 1994: 151-163

Running example: Co-evolving CS, Corr, AS

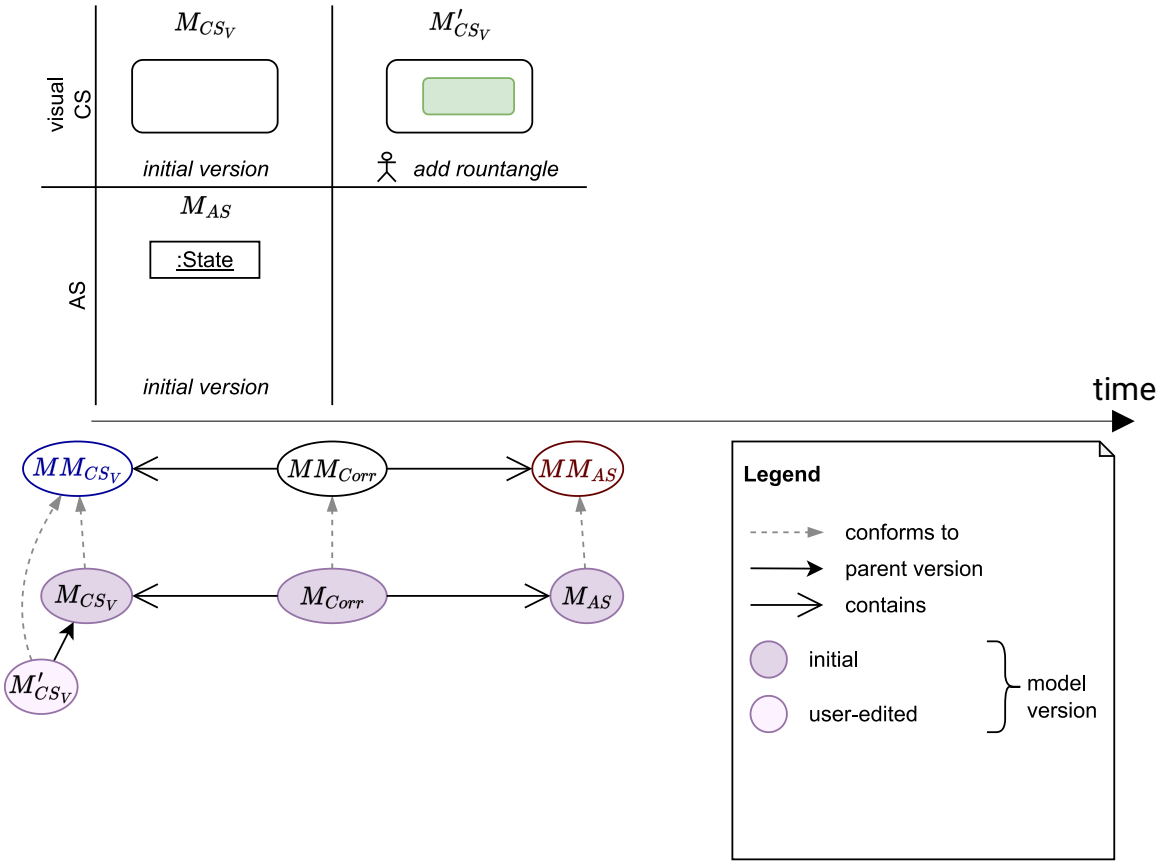
1. Initial versions of CS and AS



Running example: Co-evolving CS, Corr, AS

1. Initial versions of CS and AS

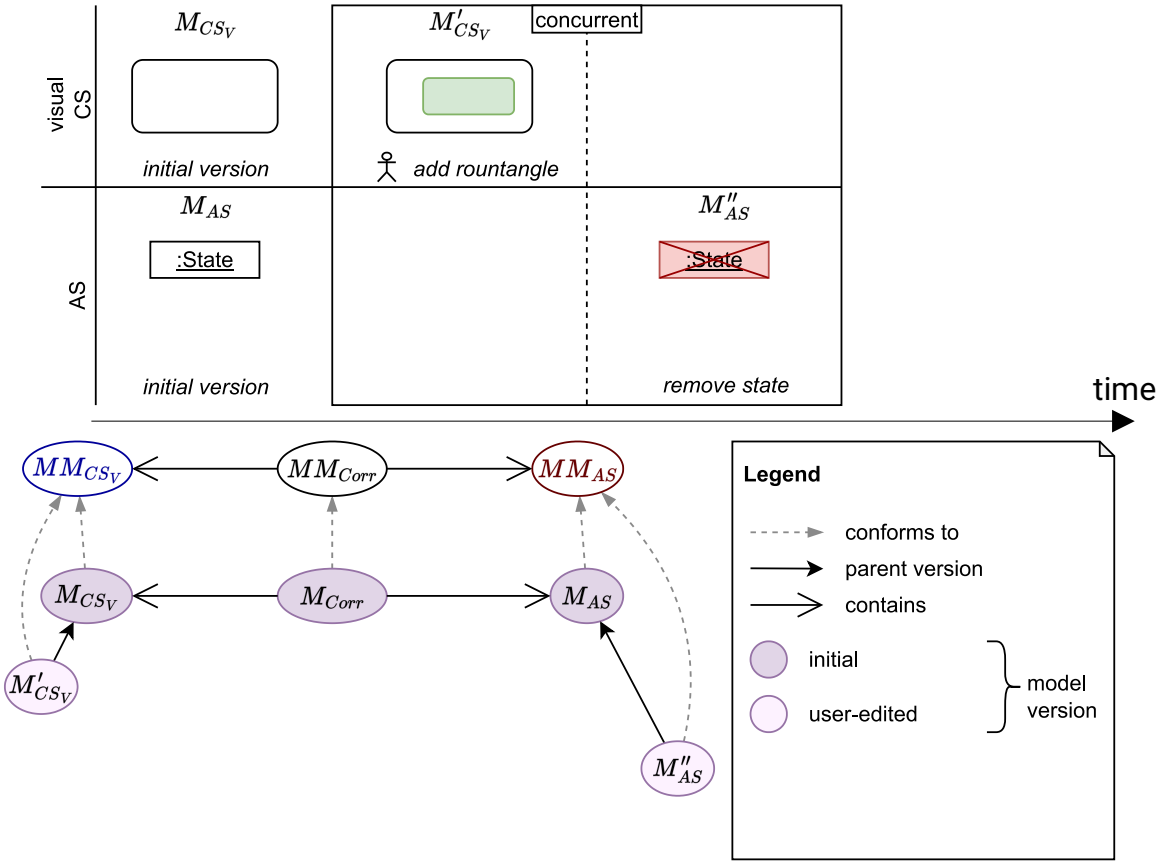
2.
   i. User change in CS creates a new CS model version



Running example: Co-evolving CS, Corr, AS

1. Initial versions of CS and AS

2. Concurrently:
   i. User change in CS creates a new CS model version
   ii. User change in AS creates a new AS model version

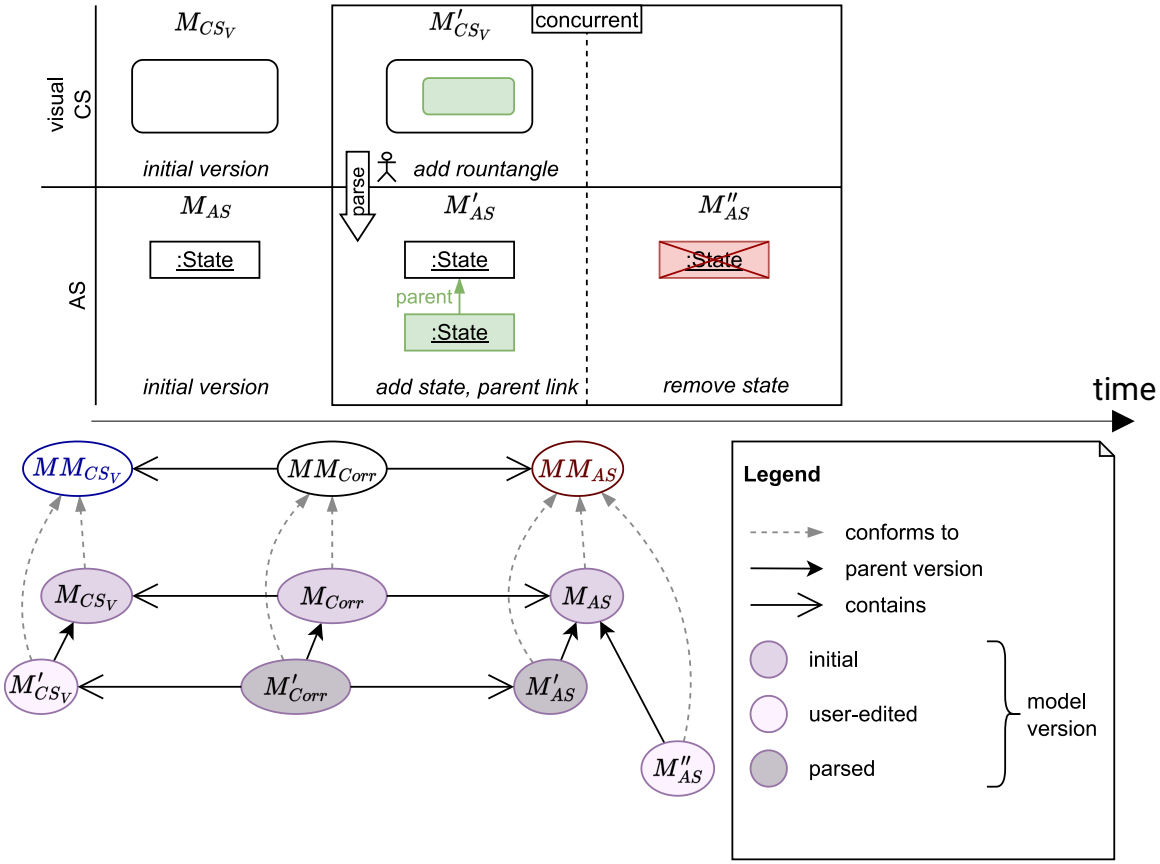


Running example: Co-evolving CS, Corr, AS

1. Initial versions of CS and AS

2. Concurrently:
   i. User change in CS creates a new CS model version
   ii. User change in AS creates a new AS model version

3.
   i. Parse changes to CS, resulting in new AS (+Corr) model version

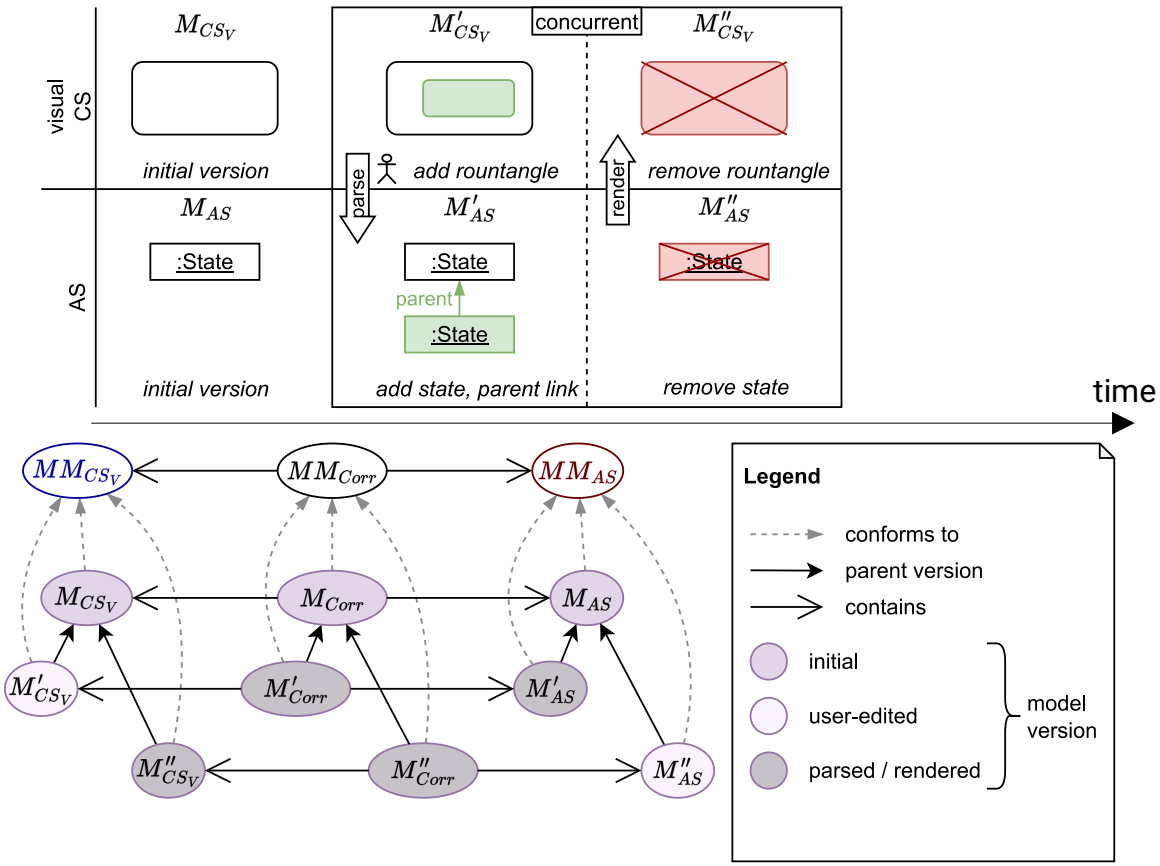


Running example: Co-evolving CS, Corr, AS

1. Initial versions of CS and AS

2. Concurrently:
   i. User change in CS creates a new CS model version
   ii. User change in AS creates a new AS model version

3.
   i. Parse changes to CS, resulting in new AS (+Corr) model version
   ii. Render changes to AS, resulting in new CS (+Corr) model version

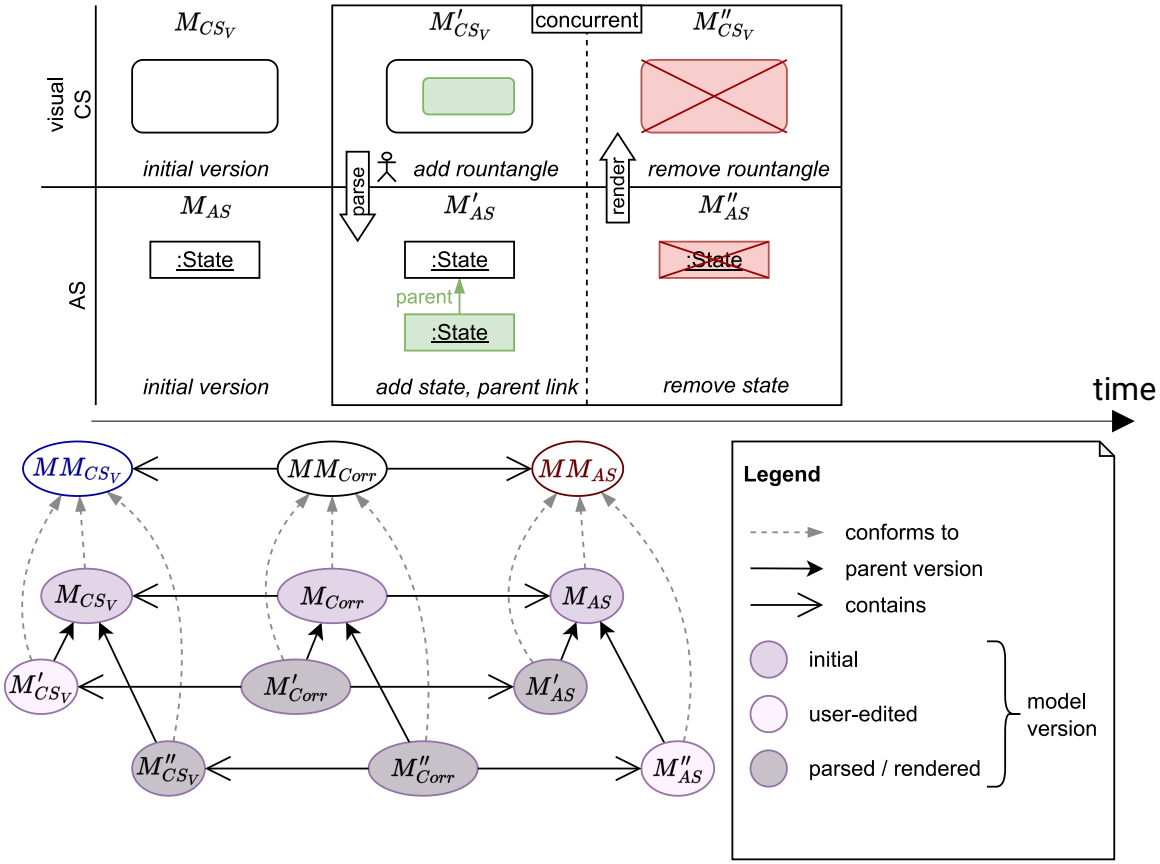


Running example: Co-evolving CS, Corr, AS

1. Initial versions of CS and AS

2. Concurrently:
   i. User change in CS creates a new CS model version
   ii. User change in AS creates a new AS model version

3. (in *any* order, at *any* time)
   i. Parse changes to CS, resulting in new AS (+Corr) model version
   ii. Render changes to AS, resulting in new CS (+Corr) model version



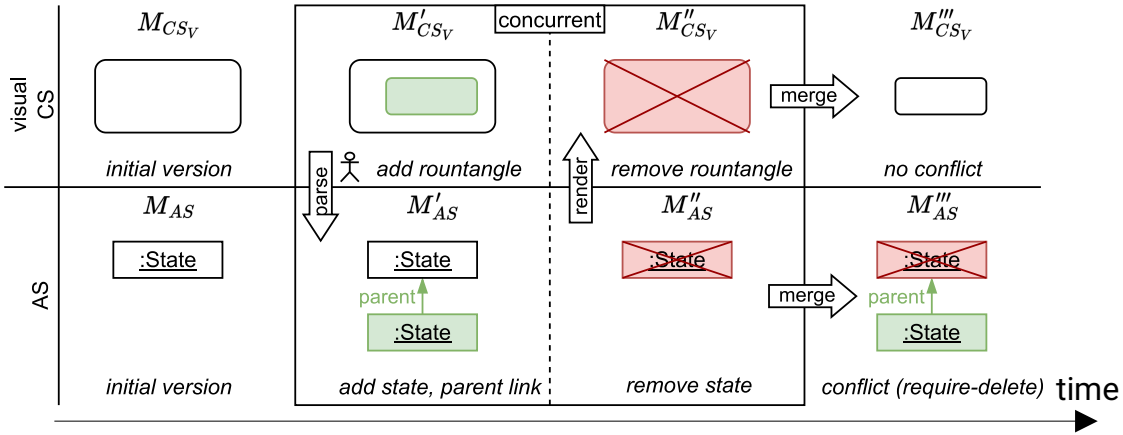
Running example: Co-evolving CS, Corr, AS

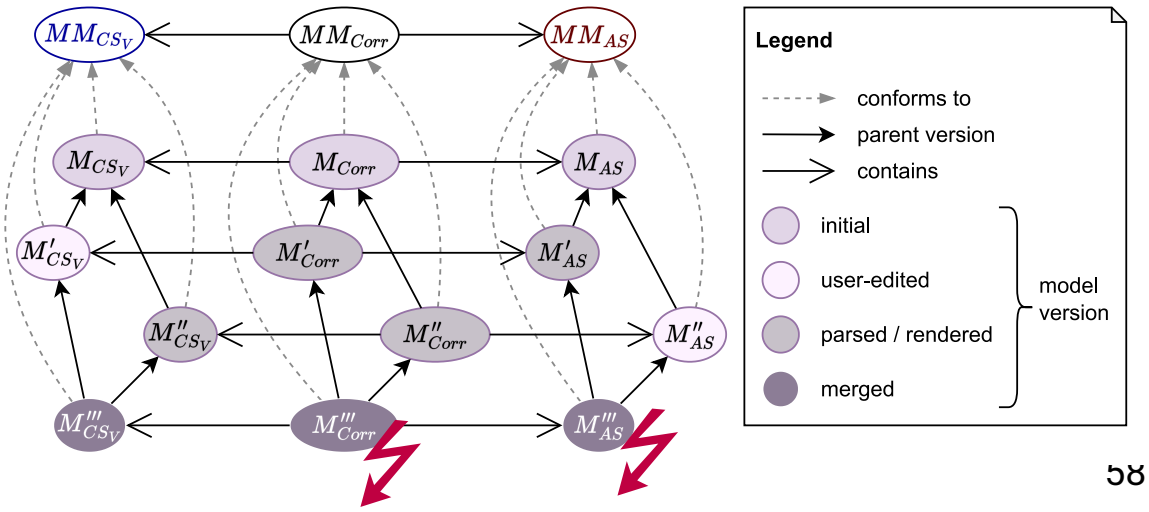
1. Initial versions of CS and AS

2. Concurrently:
   i. User change in CS creates a new CS model version
   ii. User change in AS creates a new AS model version

3. (in *any* order, at *any* time)
   i. Parse changes to CS, resulting in new AS (+Corr) model version
   ii. Render changes to AS, resulting in new CS (+Corr) model version

4. (Attempt to) merge concurrent versions at level of CS, AS, Corr





To be addressed

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

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 layout continuity
  - to deal with concurrency
  - for performance
- Fits nicely with **operation-based versioning**

How do we apply operation-based versioning?

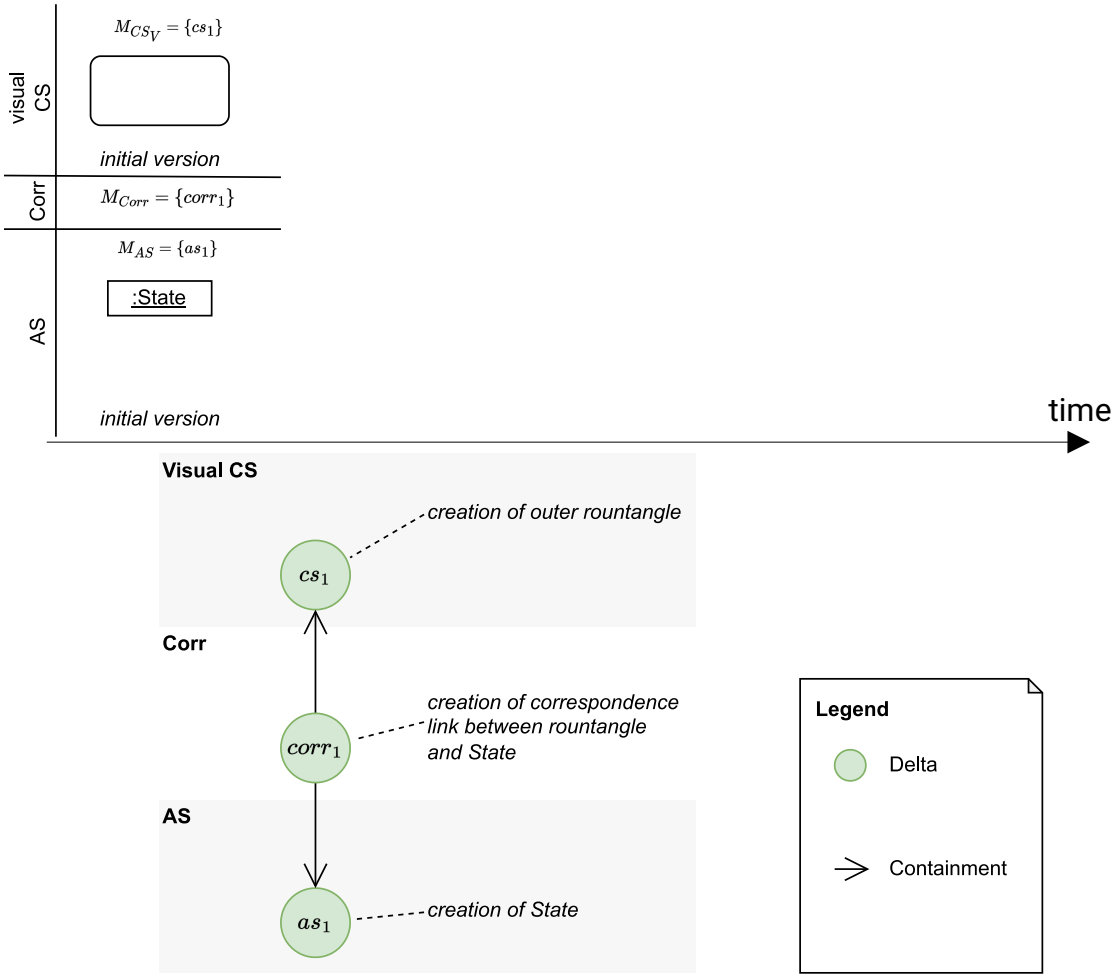
- A **delta** = a **set** of changes on a graph, caused by
  - an **edit operation** (by user)
  - a **propagated change** (by parser/renderer)
- A model **version** is just a set of deltas
  - empty version => empty graph
- A delta may **depend on** other (earlier) deltas
  - dependency types: require, update, delete
  - dependencies form a partial ordering relation on deltas
- Deltas can only be **conflicting** if they have **overlapping dependencies**
  - conflict types: update/update, update/delete, delete/require, delete/delete
  - for every new delta, can **compute** its dependencies and conflicts in (near-)**constant time**

Demo...

- https://mstro.duckdns.org/public/onion/

Running example: Persisting deltas and dependencies

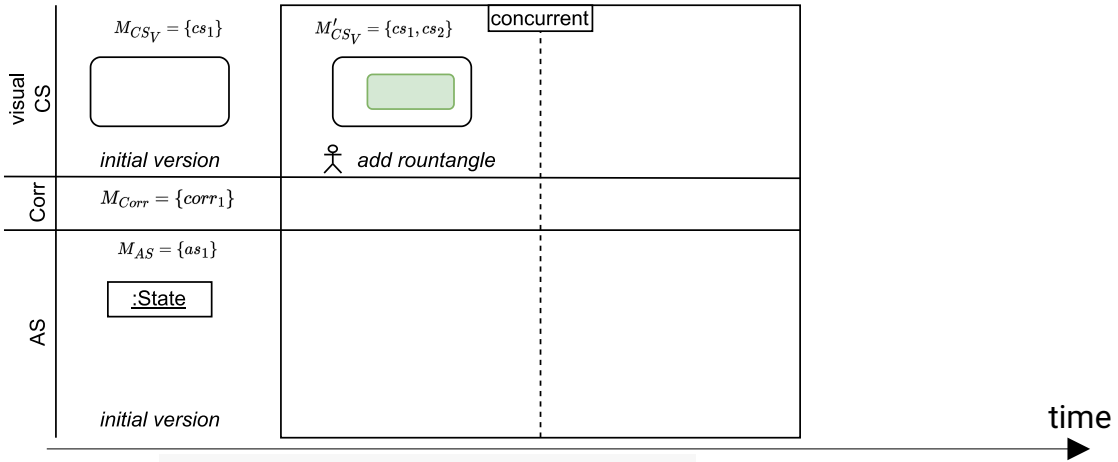
1. Initial versions assumed to each consist of a single delta

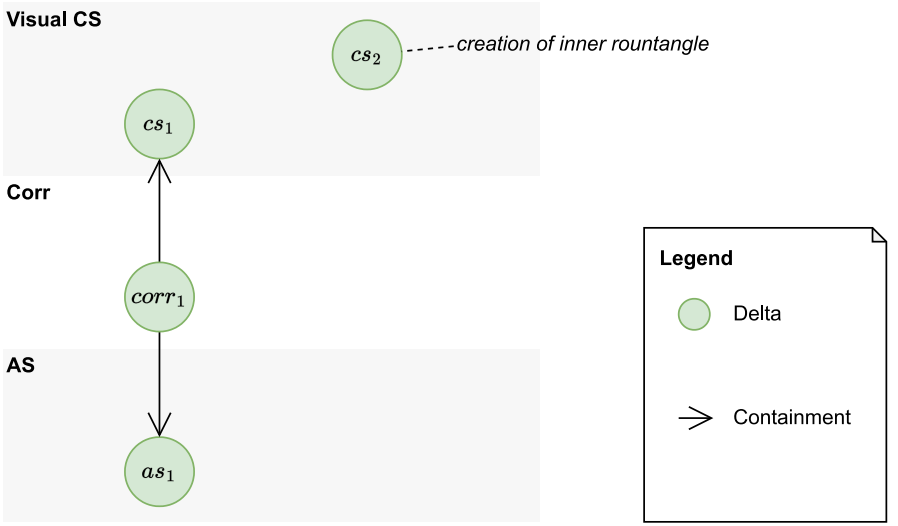


Running example: Persisting deltas and dependencies

1. Initial versions assumed to each consist of a single delta

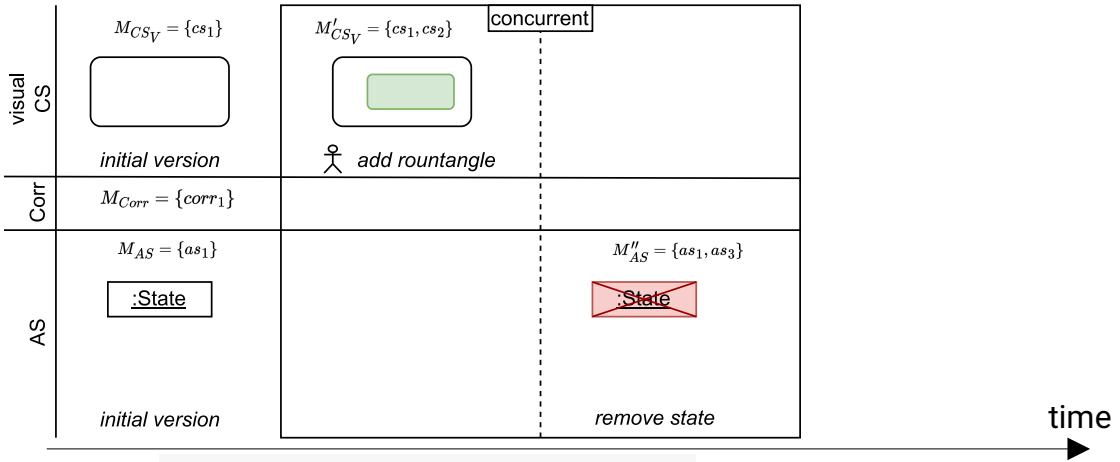
2. CS: Creation of inner rountangle does not depend on any other delta

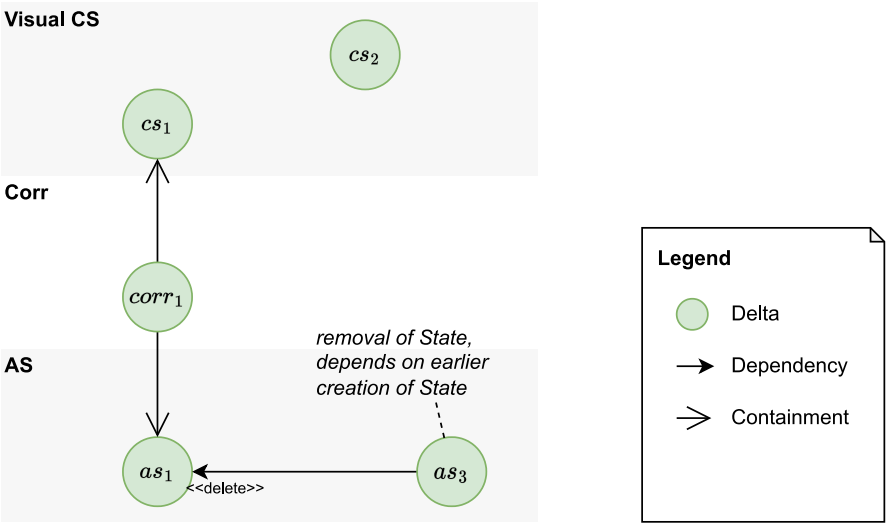




Running example: Persisting deltas and dependencies

1. Initial versions assumed to each consist of a single delta
2. CS: Creation of inner rountangle does not depend on any other delta
3. AS: Removal of State depends on earlier creation of State





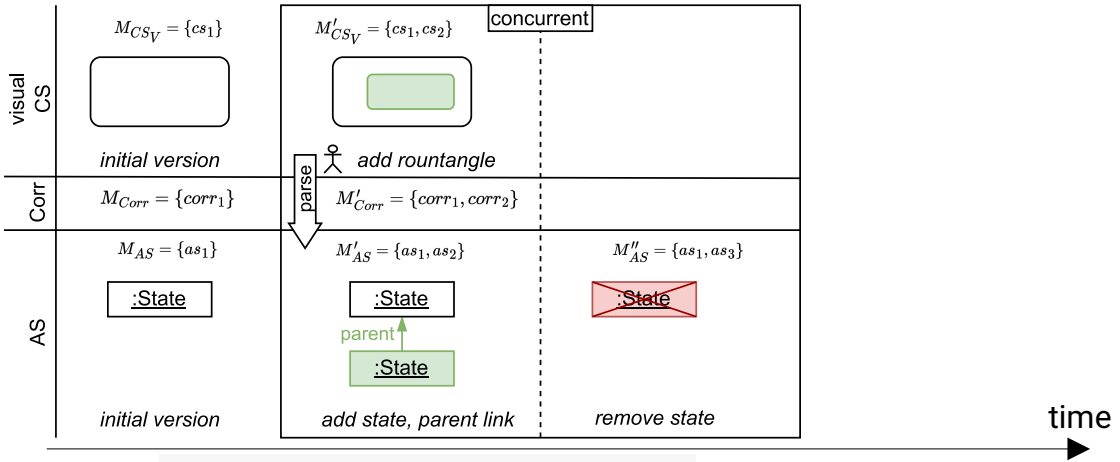
Running example: Persisting deltas and dependencies

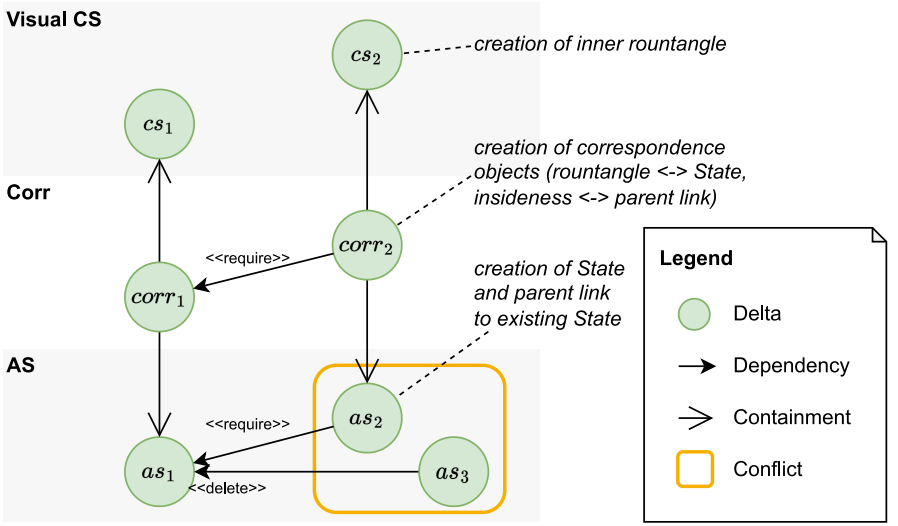
1. Initial versions assumed to each consist of a single delta

2. CS: Creation of inner rountangle does not depend on any other delta

3. AS: Removal of State depends on earlier creation of State

4. Parsing of inner rountangle creation results in 1 new delta in AS and 1 new delta in Corr





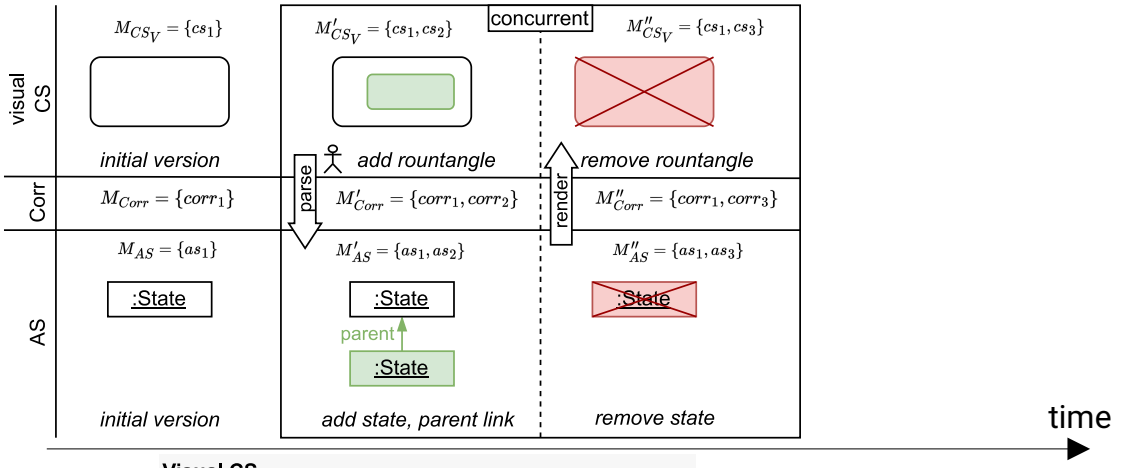
Running example: Persisting deltas and dependencies

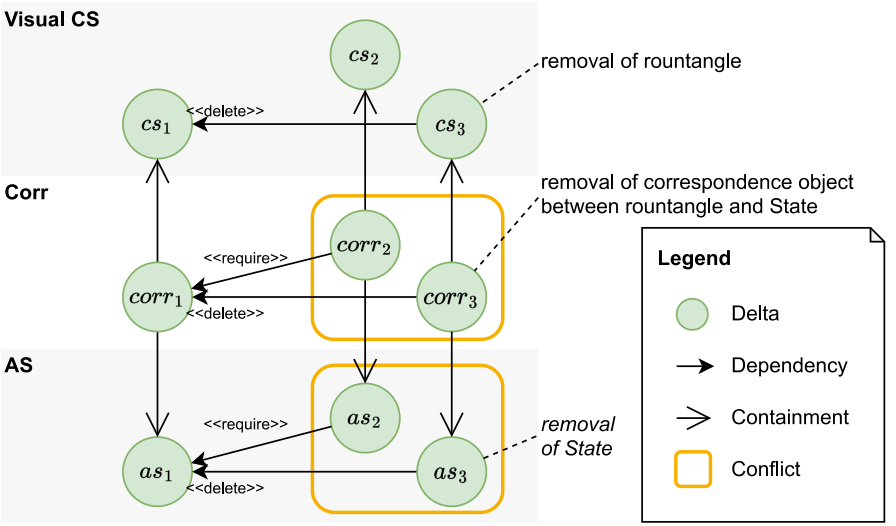
2. CS: Creation of inner rountangle does not depend on any other delta

3. AS: Removal of State depends on earlier creation of State

4. Parsing of inner rountangle creation results in 1 new delta in AS and 1 new delta in Corr

5. Rendering of State removal results in 1 new delta in CS and 1 new delta in Corr





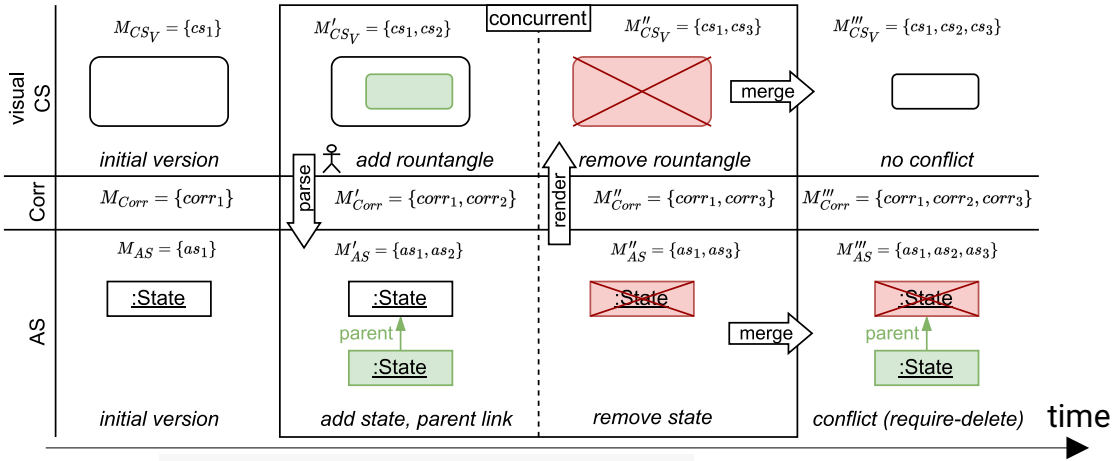
Running example: Persisting deltas and dependencies

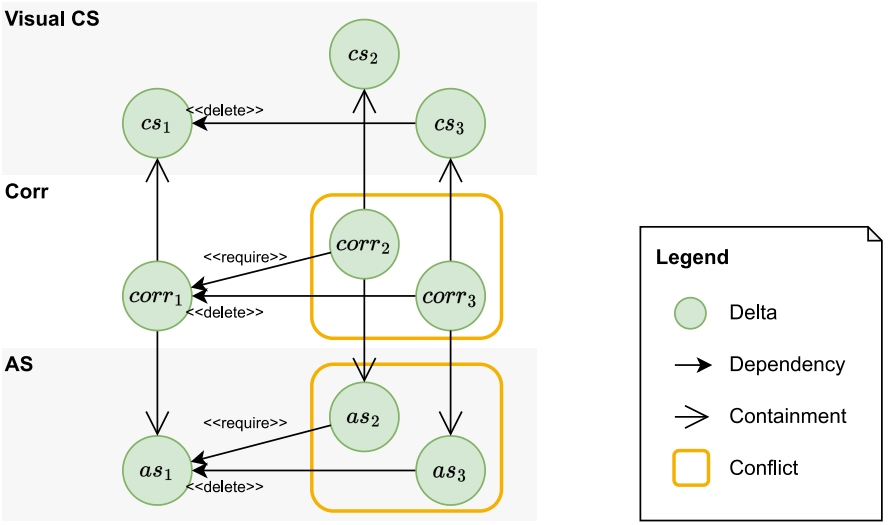
3. AS: Removal of State depends on earlier creation of State

4. Parsing of inner rountangle creation results in 1 new delta in AS and 1 new delta in Corr

5. Rendering of State removal results in 1 new delta in CS and 1 new delta in Corr

6. Merging at all levels does not result in new deltas: the merged versions simply consist of the unions of the existing deltas





Links

- Online demo
  - https://mstro.duckdns.org/public/onion/
- Source code
  - https://msdl.uantwerpen.be/git/jexelmans/onioncollab
- Publications
  - Joeri Exelmans, Jakob Pietron, Alexander Raschke, Hans Vangheluwe, Matthias Tichy:
    Optimistic Versioning for Conflict-tolerant Collaborative Blended Modeling.
    STAF Workshops 2022
    - Workshop article, original vision (still pretty consistent with implementation)
    - Was written before we started our implementation

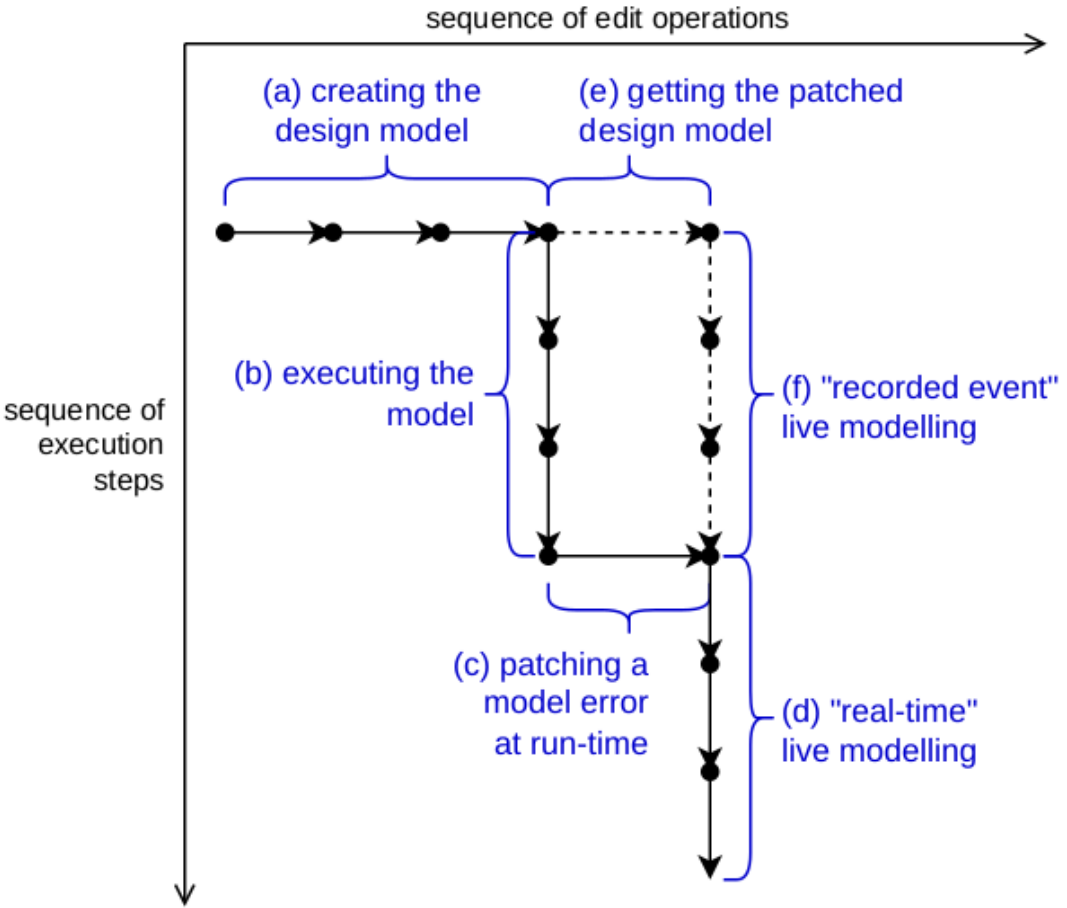
  - Joeri Exelmans, Jakob Pietron, Alexander Raschke, Hans Vangheluwe, Matthias Tichy:
    A New Versioning Approach for Collaboration in Blended Modeling.
    Journal of Computer Languages (June 2023)
    - Journal article, much more detailed, explains implementation

Part 2: Collaborative Live Modeling

Background: Live Modeling

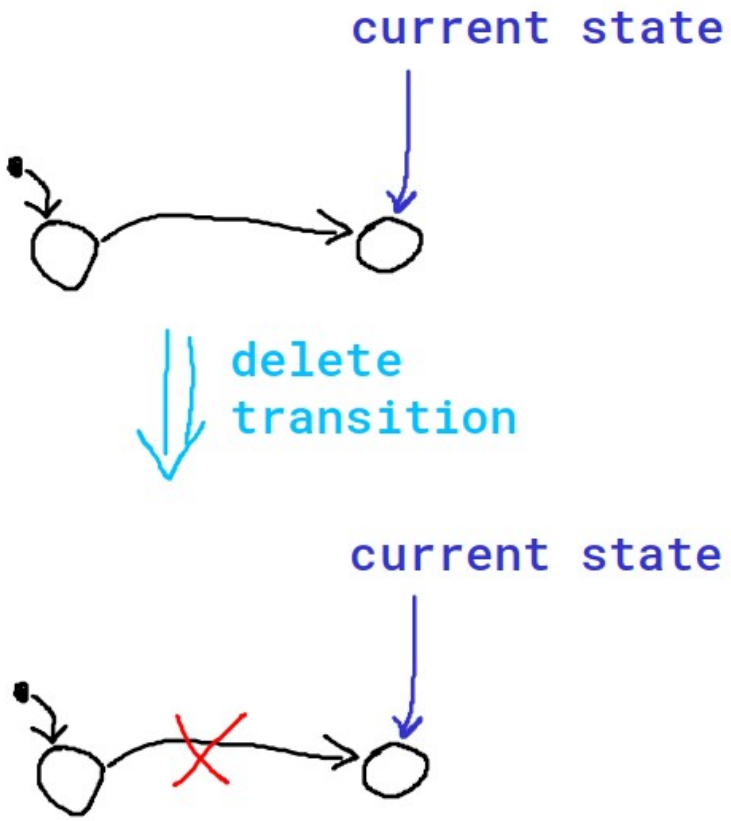
- Live Modeling = Editing a model while executing it
- Why?
  1) build model while executing it to reduce **cognitive gap**
  2) during model execution, realize your model contains a mistake and you **don’t want to restart** execution from scratch

Background: Real-time vs. recorded event live modeling[1]



[1] Yentl Van Tendeloo, Simon Van Mierlo, Hans Vangheluwe:
A Multi-Paradigm Modelling approach to live modelling. Softw. Syst. Model. 18(5): 2821-2842 (2019)

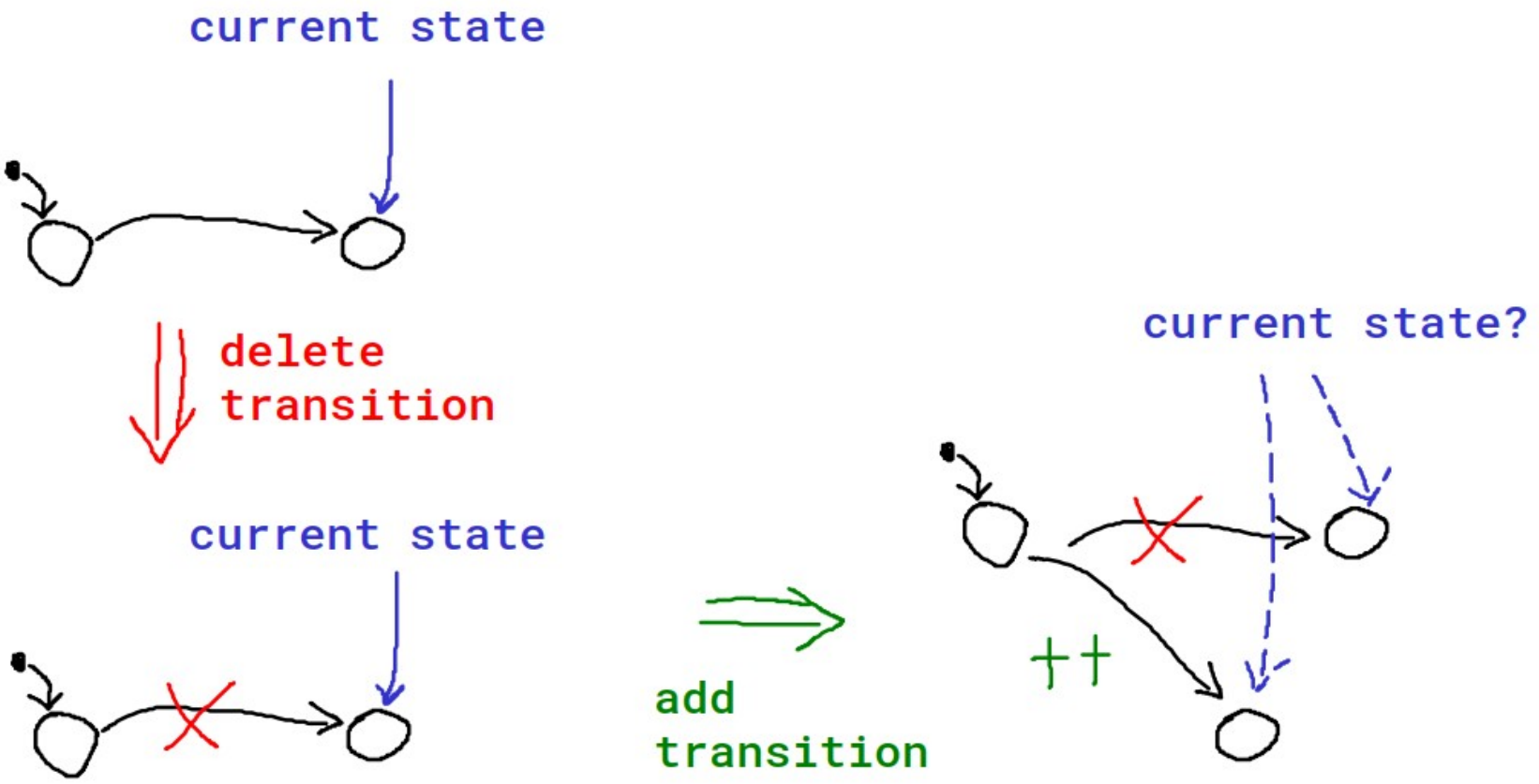
Background: Real-time vs. recorded event live modeling



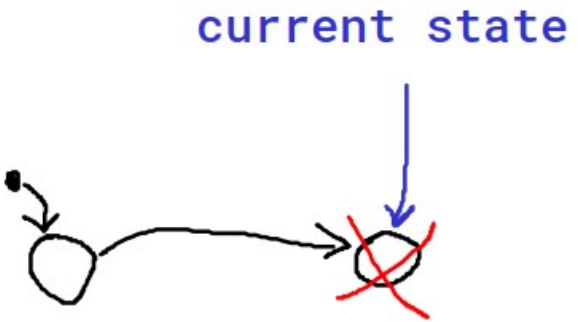
recorded event LM:
cannot replay execution

real-time LM:
no problem, just continue execution

Background: Real-time vs. recorded event live modeling



Background: Real-time live modeling



- State automata example: What if **current state** is **deleted**?
  - **forbid** deletion?
  - **ask user** for new current state?
  - set current state back to **initial state**?

Background: Real-time vs. recorded event live modeling

- Recorded event
  - **restarts** execution and automatically **replays** all execution steps (if possible)
  - cannot end up in unreachable run-time configuration
- Real-time
  - “patch”/consolidate ongoing execution with updated design model
  - **most intuitive** to user
  - could end up in unreachable run-time configuration

Real-time + notification when in unreachable configuration == best of both worlds?

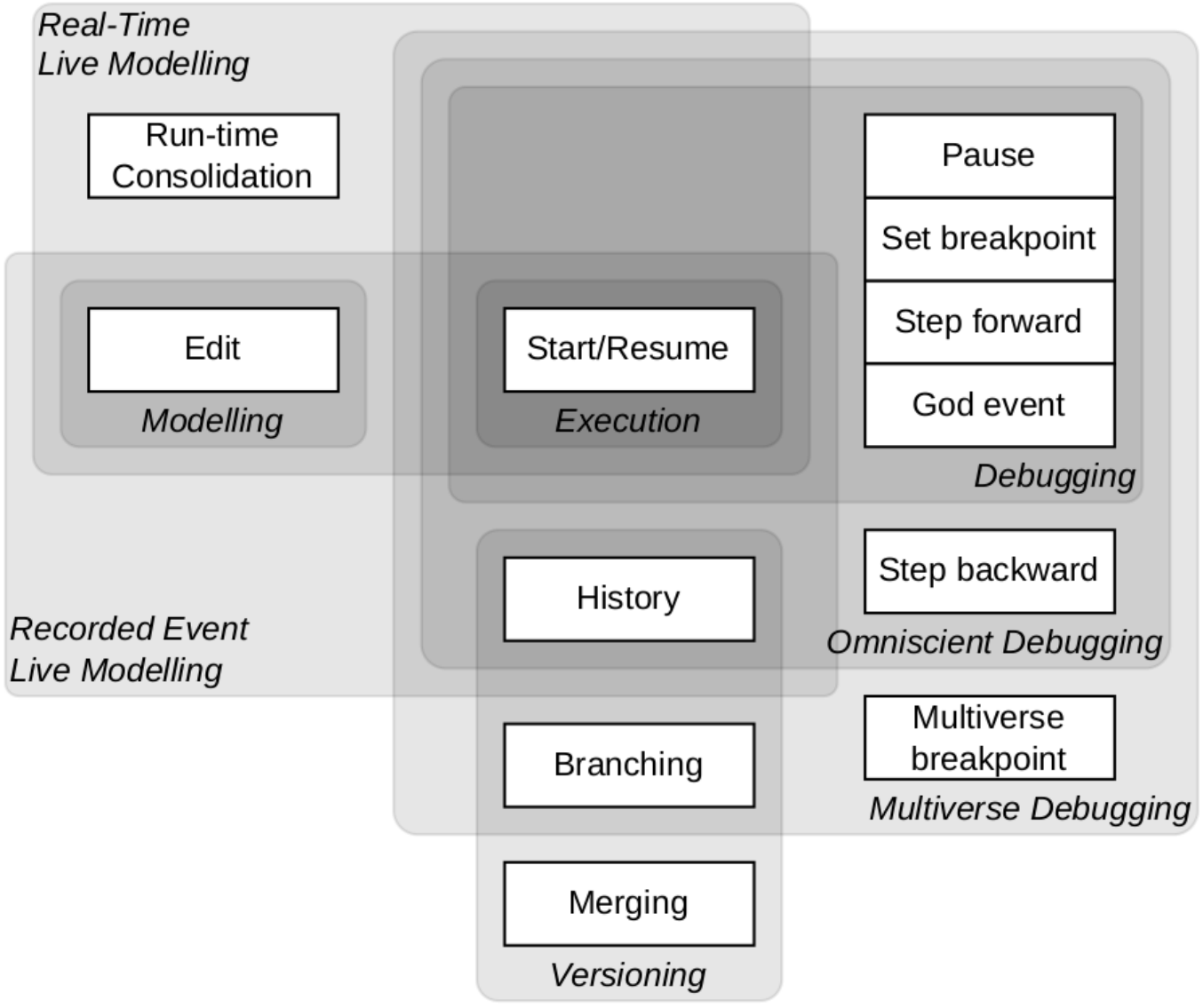
Background: Omniscient and multi-verse debugging

- Omniscient debugging[1]:
  - ability to **step back** in time
  - history remains linear (like undo)
- Multiverse debugging[2]:
  - additional ability to **jump** accross different branches of execution
  - history is branching
  - useful for **non-deterministic** languages
  - “multiverse” breakpoint (run until condition satisfied in *some branch)*

[1] Guillaume Pothier, Éric Tanter: **Back to the Future: Omniscient Debugging**. IEEE Softw. 26(6): 78-85 (2009)

[2] Matthias Pasquier, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, Luka Le Roux, Loïc Lagadec:
Practical multiverse debugging through user-defined reductions: application to UML models. MoDELS 2022: 87-97

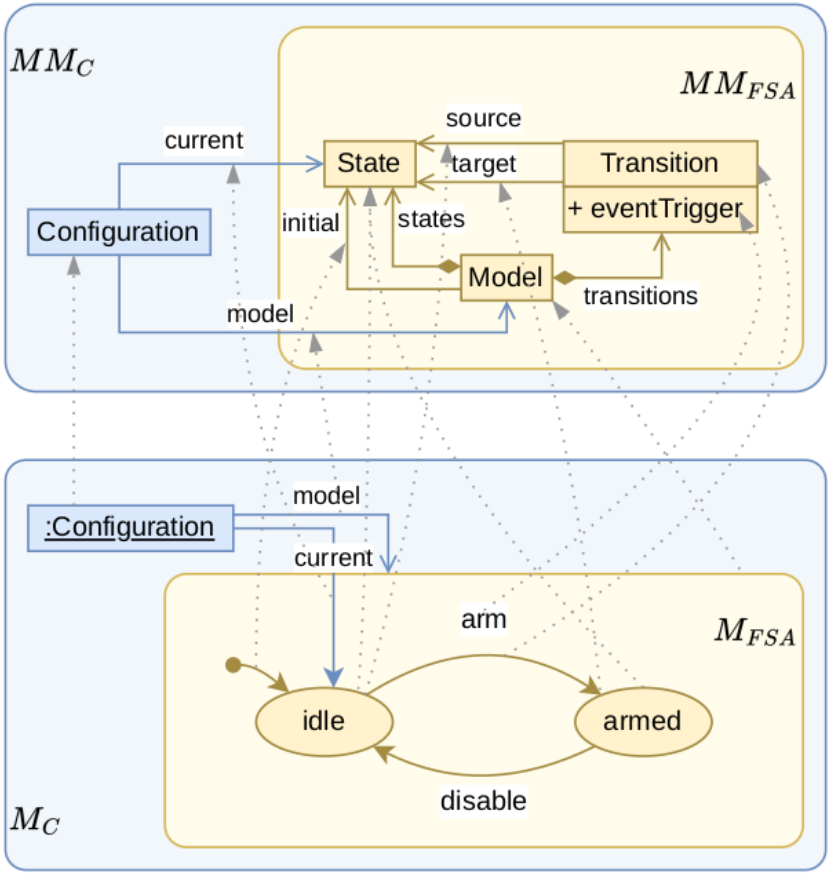
Big Picture


Real-Time
Live Modelling
Run-time
Consolidation
Edit
Modelling
Start/Resume
Execution
Pause
Set breakpoint
Step forward
God event
Debugging
Recorded Event
Live Modelling
History
Step backward
Omniscient Debugging
Branching
Multiverse
breakpoint
Multiverse Debugging
Merging
Versioning

Solution

Design model
= what you edit

Run-time model
= current (stable*) execution state
(of interpreter)
reads and points at elements
of design model



* stable meaning: in between
macro-steps (**not** micro-steps)

Solution

- We use **operation-based versioning** to record changes on design model and run-time model
  - e.g., edit operation → new version of design model
  - e.g., execution step → new version of run-time model
  - e.g., “god events” → new version of run-time model
- We record **dependencies** between deltas
- Run-time model **embeds** design model

  (just like correspondence model embeds CS and AS models)
  - run-time deltas can depend on design deltas (but not the other way around)
  - conflict between design delta and run-time delta

    = cannot combine some execution step / god event with some edit operation

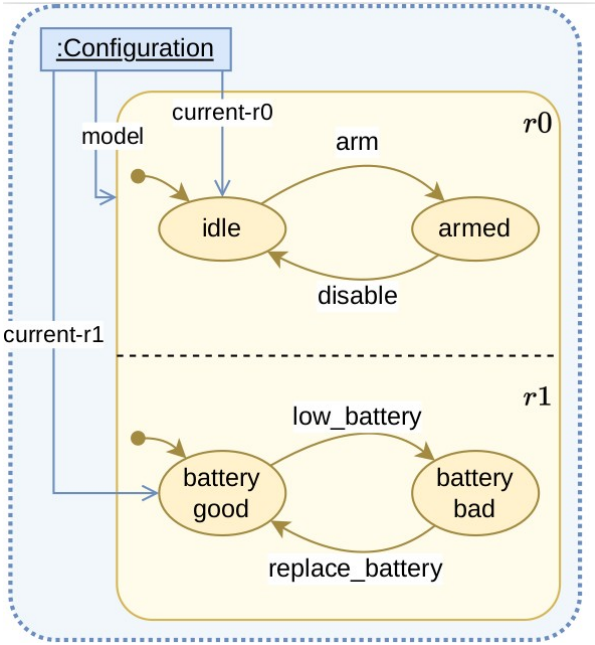
    => impossible configuration

Demo...

- https://mstro.duckdns.org/public/onion/

Additional benefits

- “Who set this variable (and when)?”
  - for every element in any model version, we already know the delta that most recently touched it
- Discovering parallel convergence
  - at run-time, independent deltas represent execution steps that can be (safely) taken in any order



Future work (ongoing here in Brest)

- Current implementation only records write-dependencies
  - Should also record the *read*-dependencies to find R/W conflicts

    => currently we fail to detect certain conflicts (*false negatives*)
- Current implementation has some ad-hoc components
  - e.g., history structure, graph encoding can made **more generic**, to ease integration with existing language implementations, model-checkers, ...
  - **De- and re-construct** as composable parts
  - Convinced that **dependencies** between deltas are **fundamental** to merging
- Creating a Statecharts + action language demo
- Also recording the interpreter's state in between micro-steps

Links

- Online demo
  - https://mstro.duckdns.org/public/onion/
- Source code
  - https://msdl.uantwerpen.be/git/jexelmans/onioncollab
- Publication
  - Joeri Exelmans, Ciprian Teodorov, Robert Heinrich, Alexander Egyed, Hans Vangheluwe:
    Collaborative Live Modelling by Language-Agnostic Versioning.
    MoDELS (Companion) 2023