Version Control in MDE

Adrian Rutle[1], Alessandro Rossini[2], Yngve Lamo[1], Uwe Wolter[2]

[1]Faculty of Engineering, Bergen University College, Norway

[2]Department of Informatics, University of Bergen, Norway

20 November 2008

HØGSKOLEN I BERGEN

Outline

Model-Driven Engineering

- Model-Driven Engineering (MDE)
  - models as primary artefacts of the software development process
- Models undergo a complex evolution during their life cycles
  - need for techniques and tools to support model evolution activities
- Version control enables concurrent and distributed software development

Version Control Paradigms

- *Lock-Modify-Unlock*
  - lock an artefact
  - modify it
  - release the lock
- *Copy-Modify-Merge*
  - create a local copy of the repository
  - modify the local copy independently
  - detect conflicting changes
  - merge the modifications into the repository

Motivation

- Lock-modify-unlock is workable if the developers always know who is planning to do what
- Traditional copy-modify-merge systems are focused on the management of text-based files
  - techniques based on per-line comparison
  - not suitable for graph-based structures
- Adoption of the copy-modify-merge paradigm enables effective version control in MDE

Motivation

- Solutions proposed by research are not sufficient to enable version control in MDE
  - lack of mature techniques targeting graph-based structures
  - same operations are given different semantics in different works/tools
  - concepts are described only semiformally
  - terminology is not precise and unique

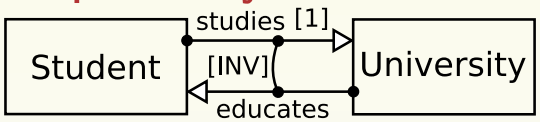
Our contribution

- Formalisation of the copy-modify-merge paradigm in MDE
- Definition of common models and merge models by means of pullback and pushout constructions respectively
- Representation of model differences and detection of syntactic conflicts by using Diagram Predicate Framework (DPF)

Outline

Example

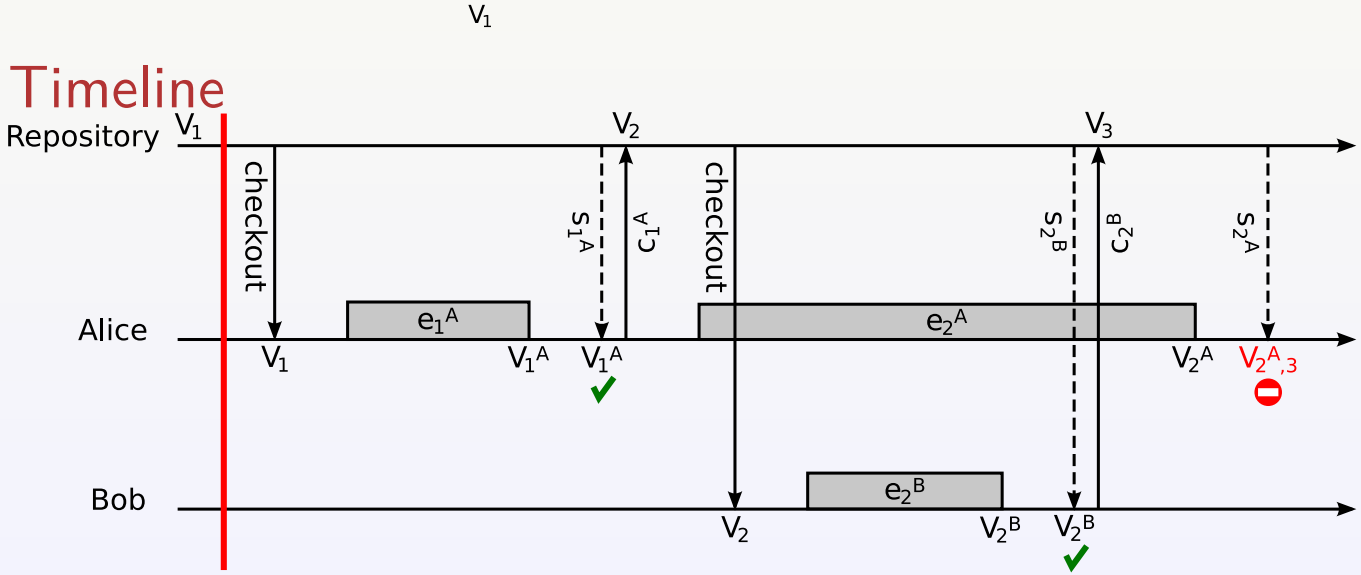
Repository



Alice's local copy

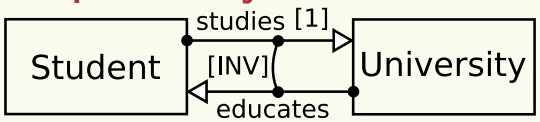
Bob's local copy

Timeline

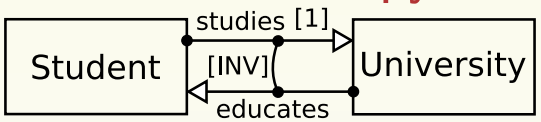


Example

Repository

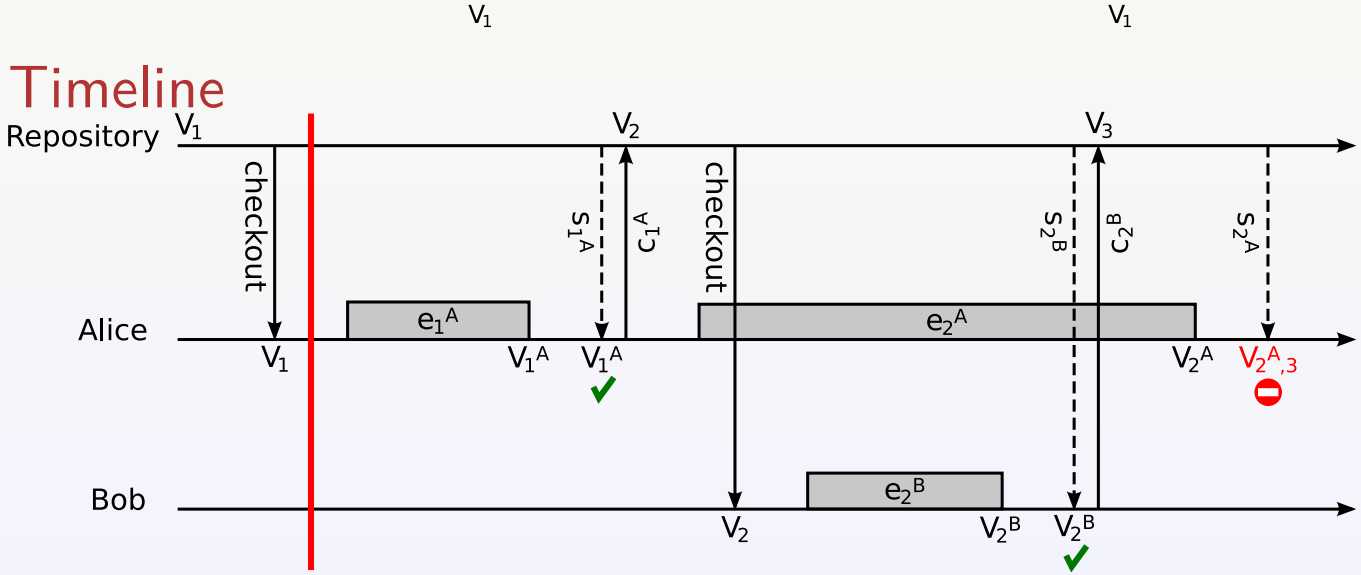


Alice's local copy



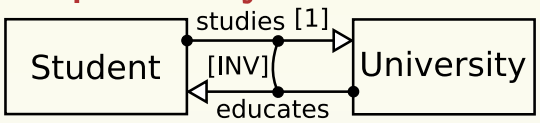
Bob's local copy

Timeline

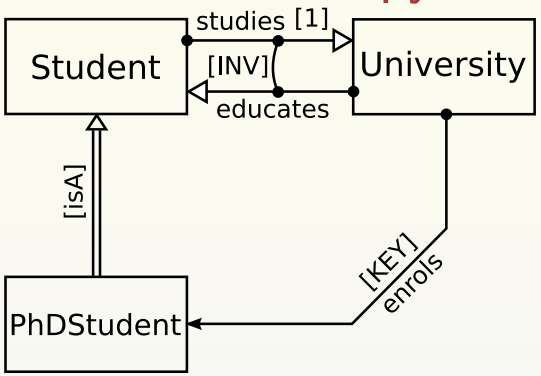


Example

Repository

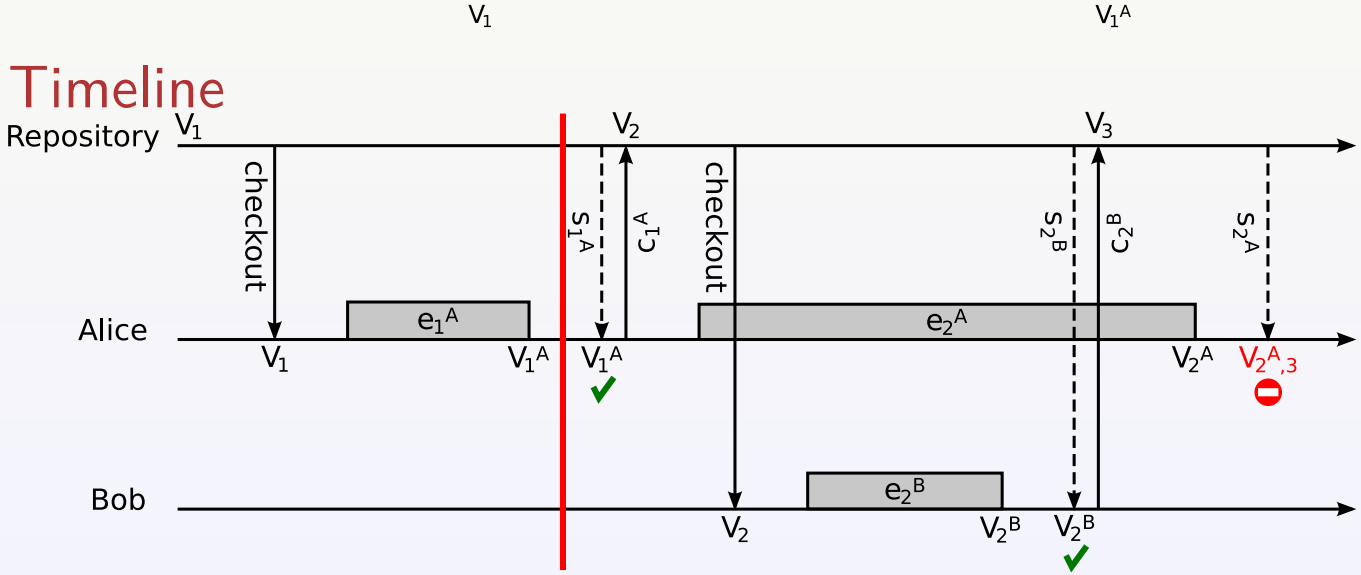


Alice's local copy



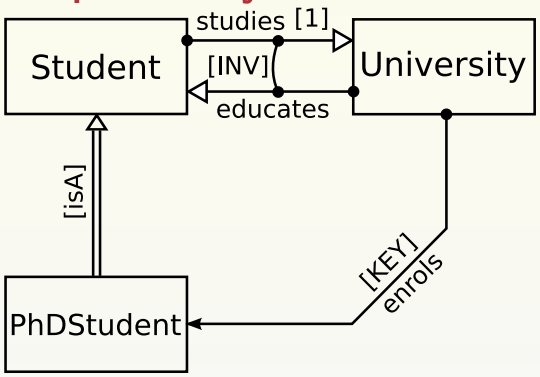
Bob's local copy

Timeline



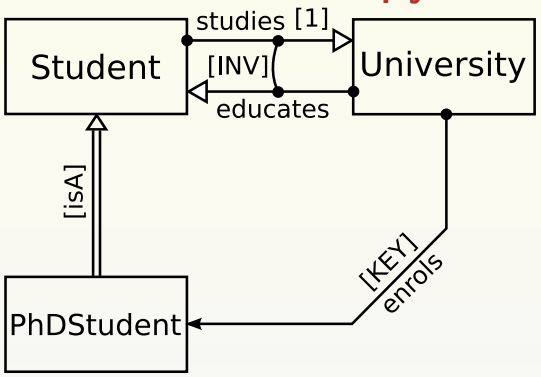
Example

Repository



V_2

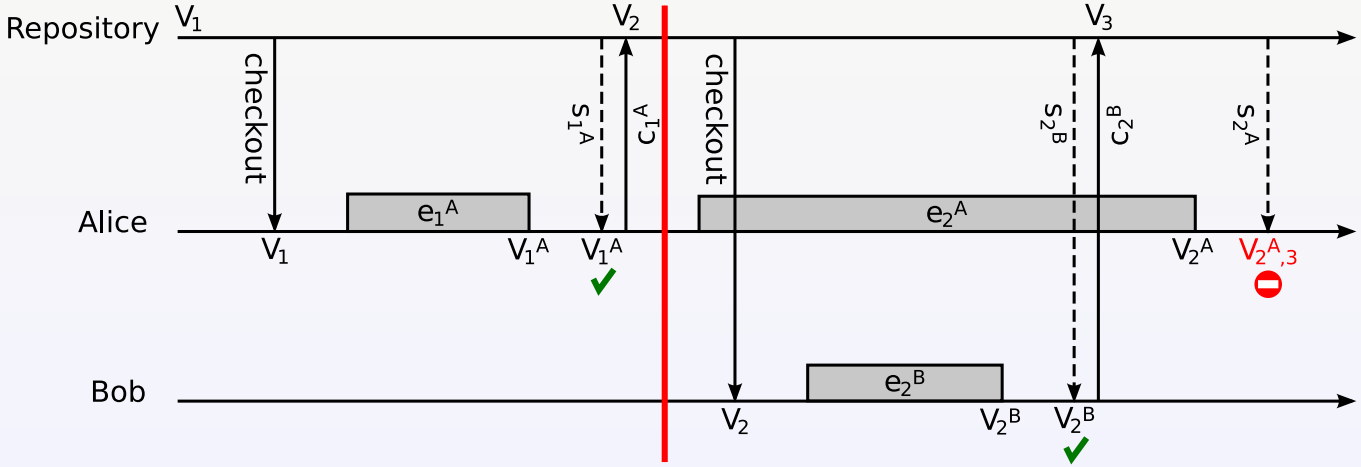
Alice's local copy



V_2

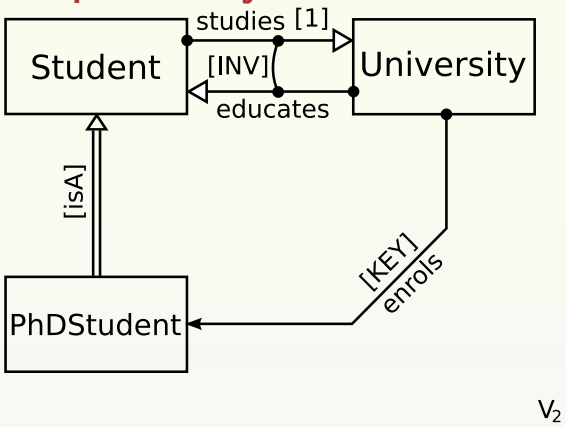
Bob's local copy

Timeline

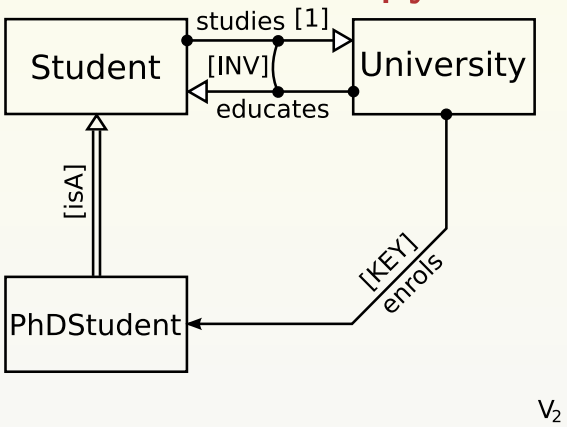


Example

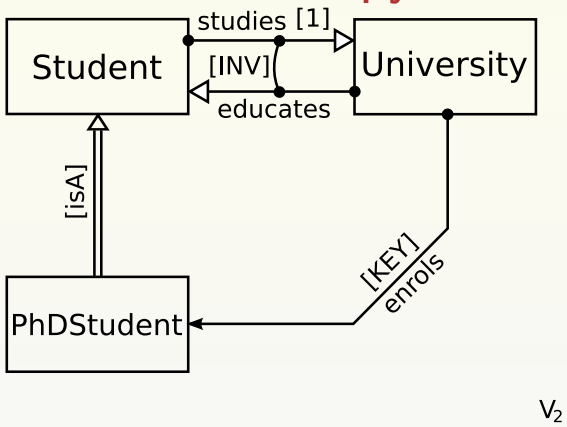
Repository



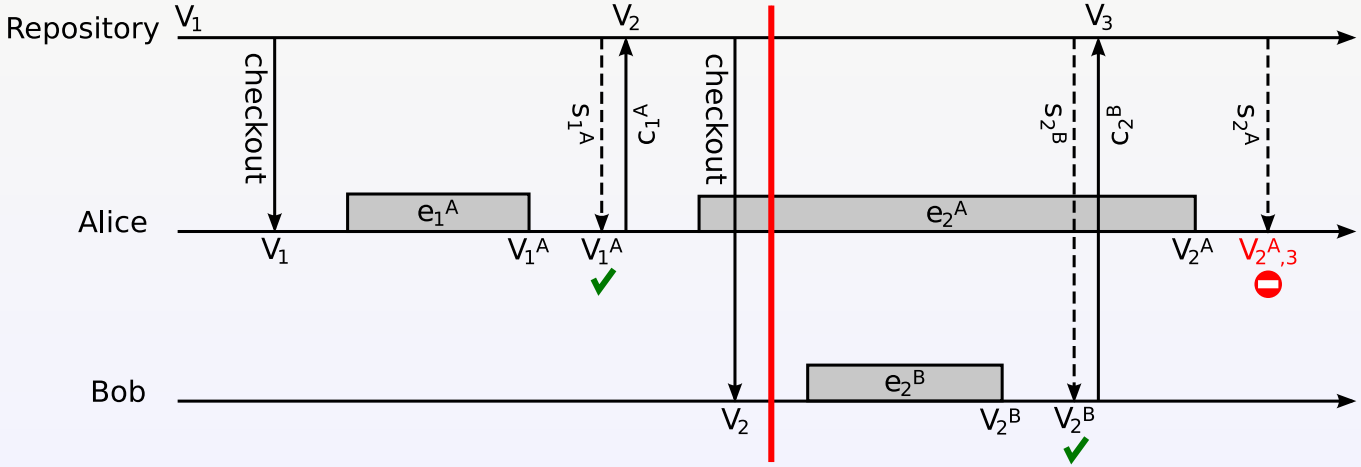
Alice's local copy



Bob's local copy

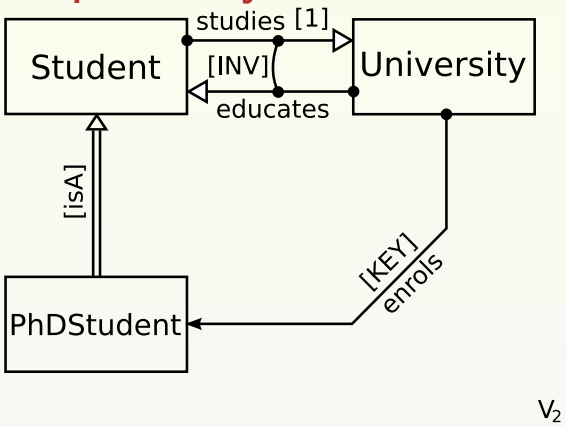


Timeline

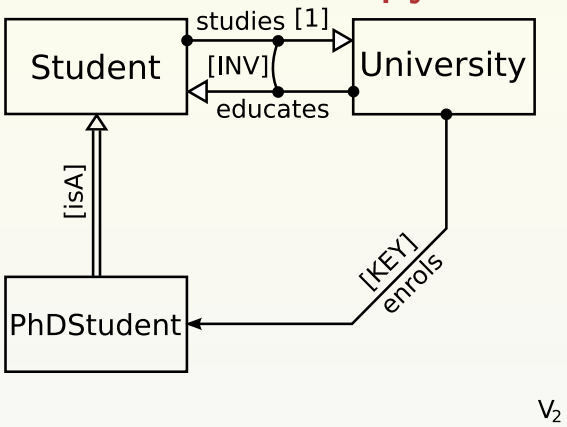


Example

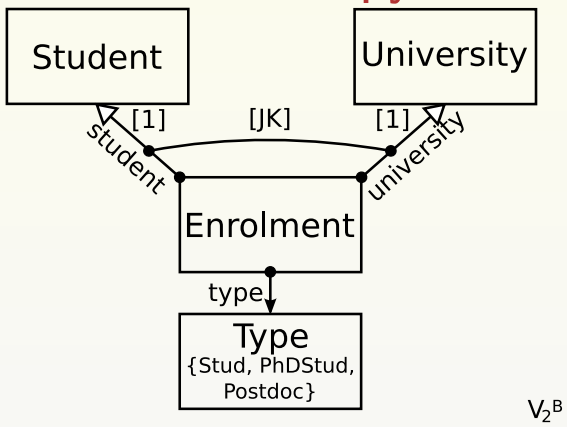
Repository



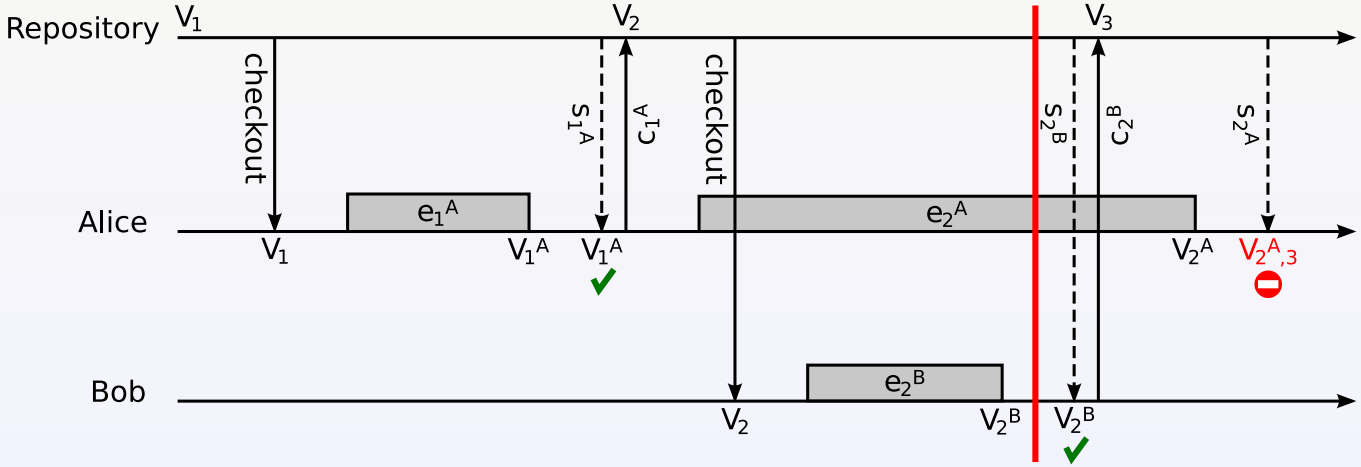
Alice's local copy



Bob's local copy

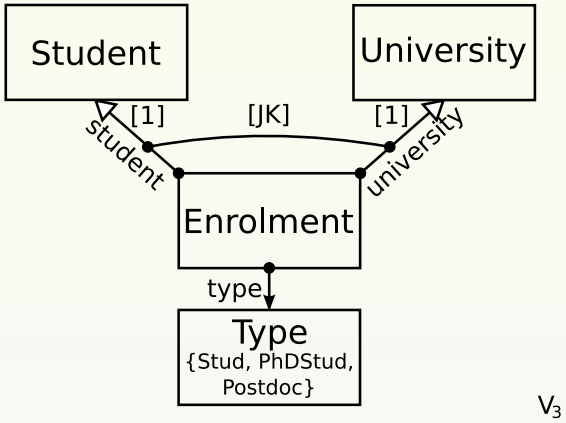


Timeline

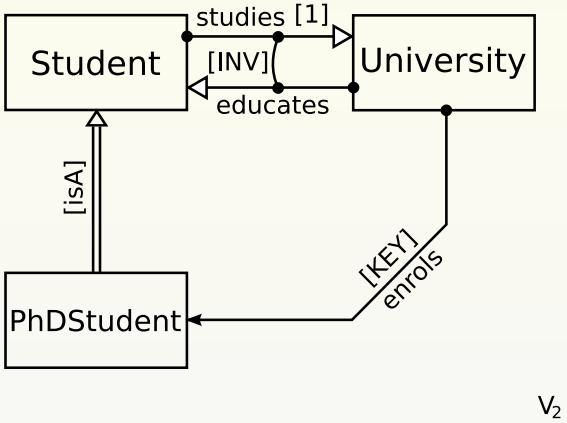


Example

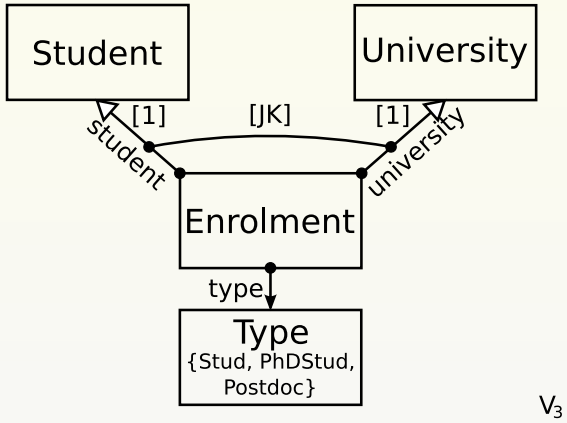
Repository



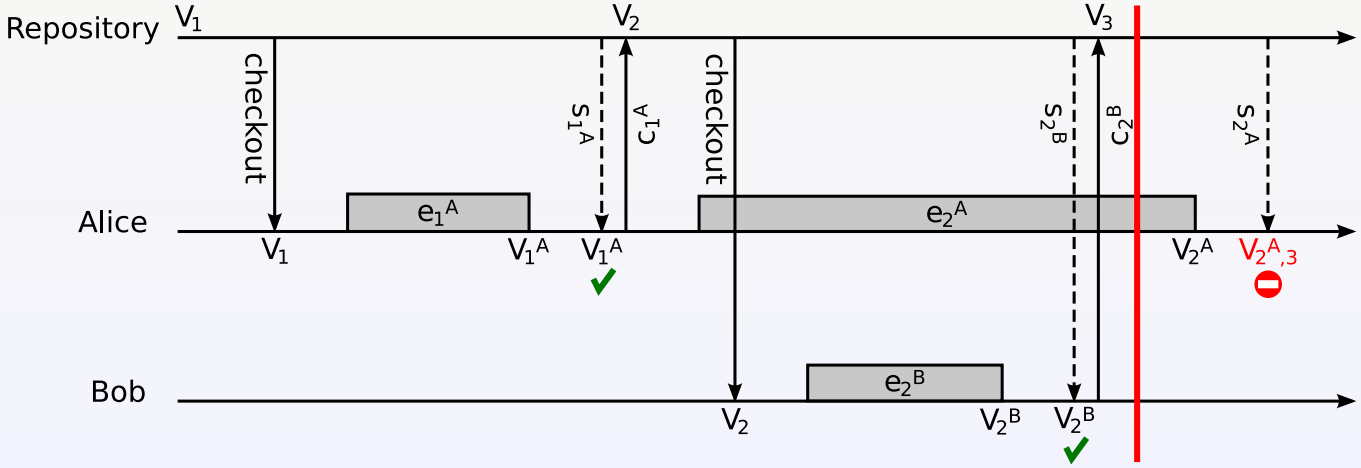
Alice's local copy



Bob's local copy

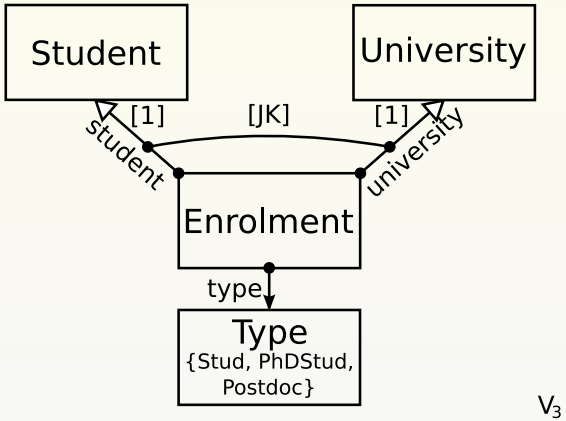


Timeline

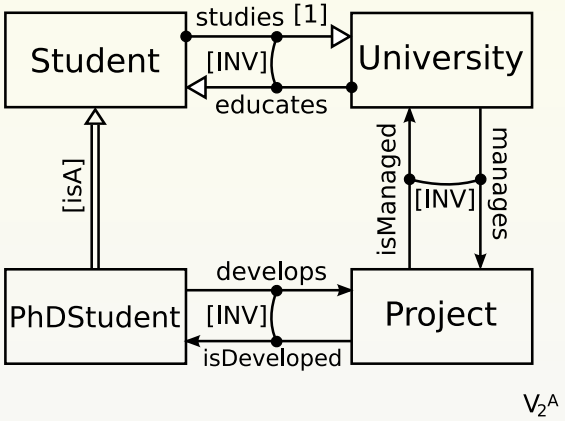


Example

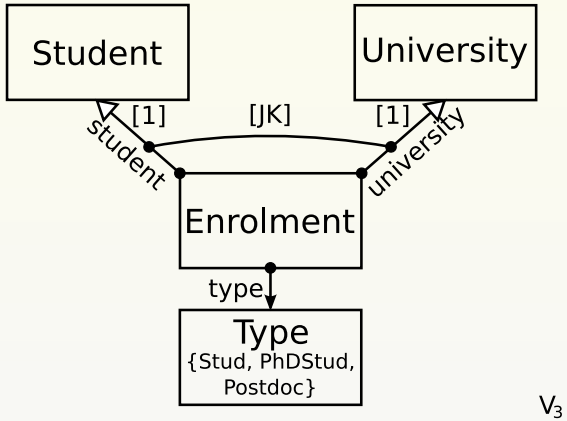
Repository



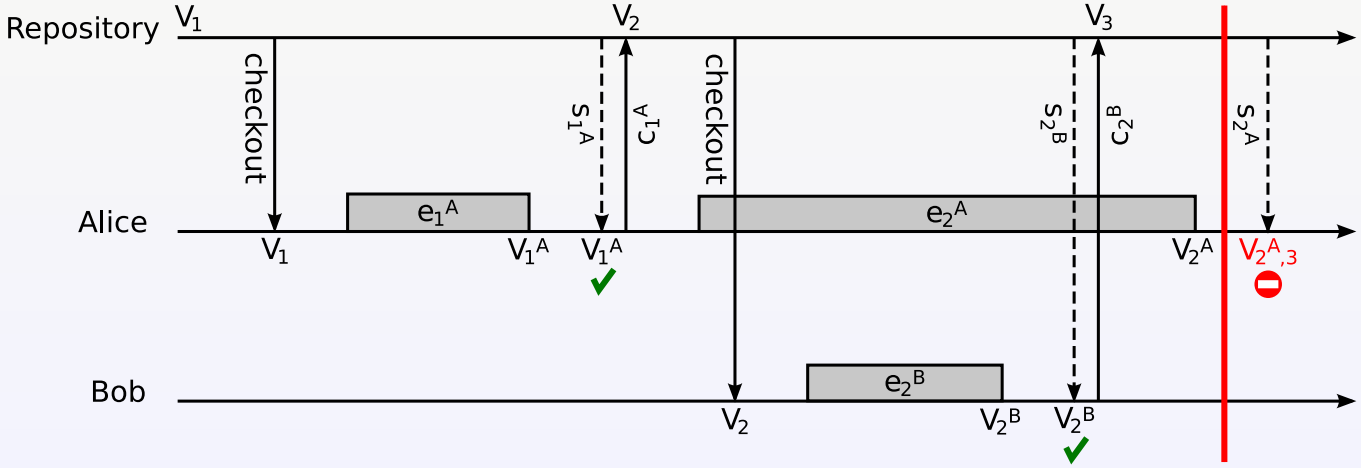
Alice's local copy



Bob's local copy

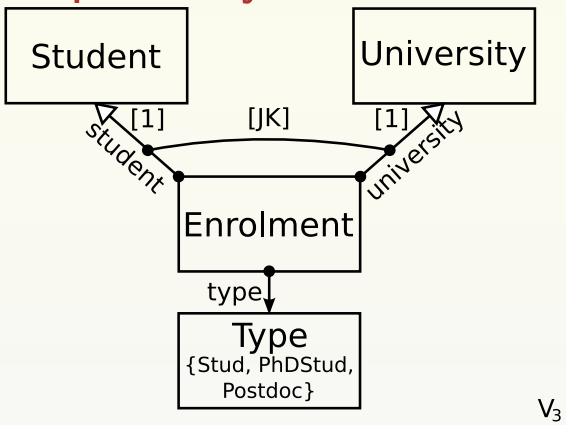


Timeline

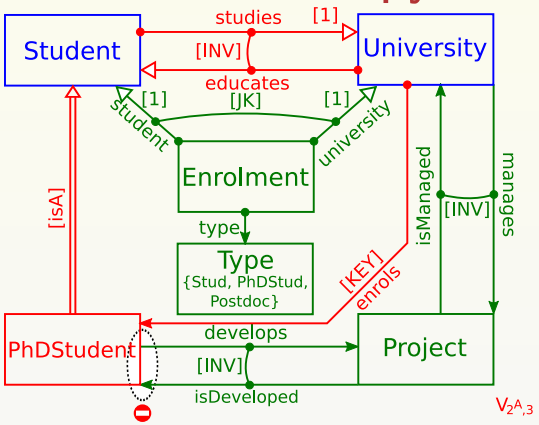


Example

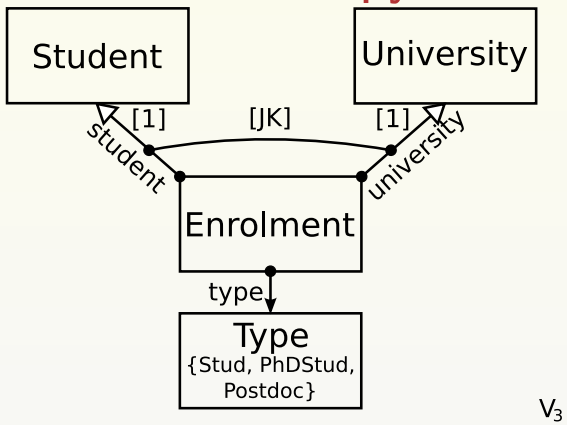
Repository



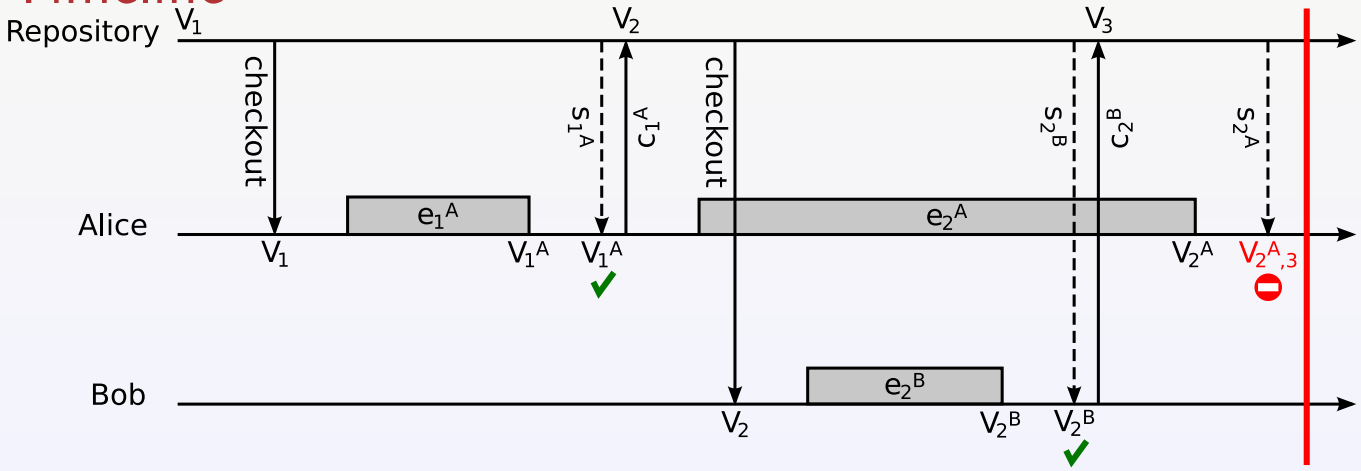
Alice's local copy



Bob's local copy



Timeline



Outline

Common Model

- The development environment must keep track over what is common between subsequent models
- Existing approaches to the identification of mappings/differences
  - *soft-linking*: structurally equivalent elements imply equal elements
  - *hard-linking*: elements with equal identifiers are seen as equal elements

Common Model

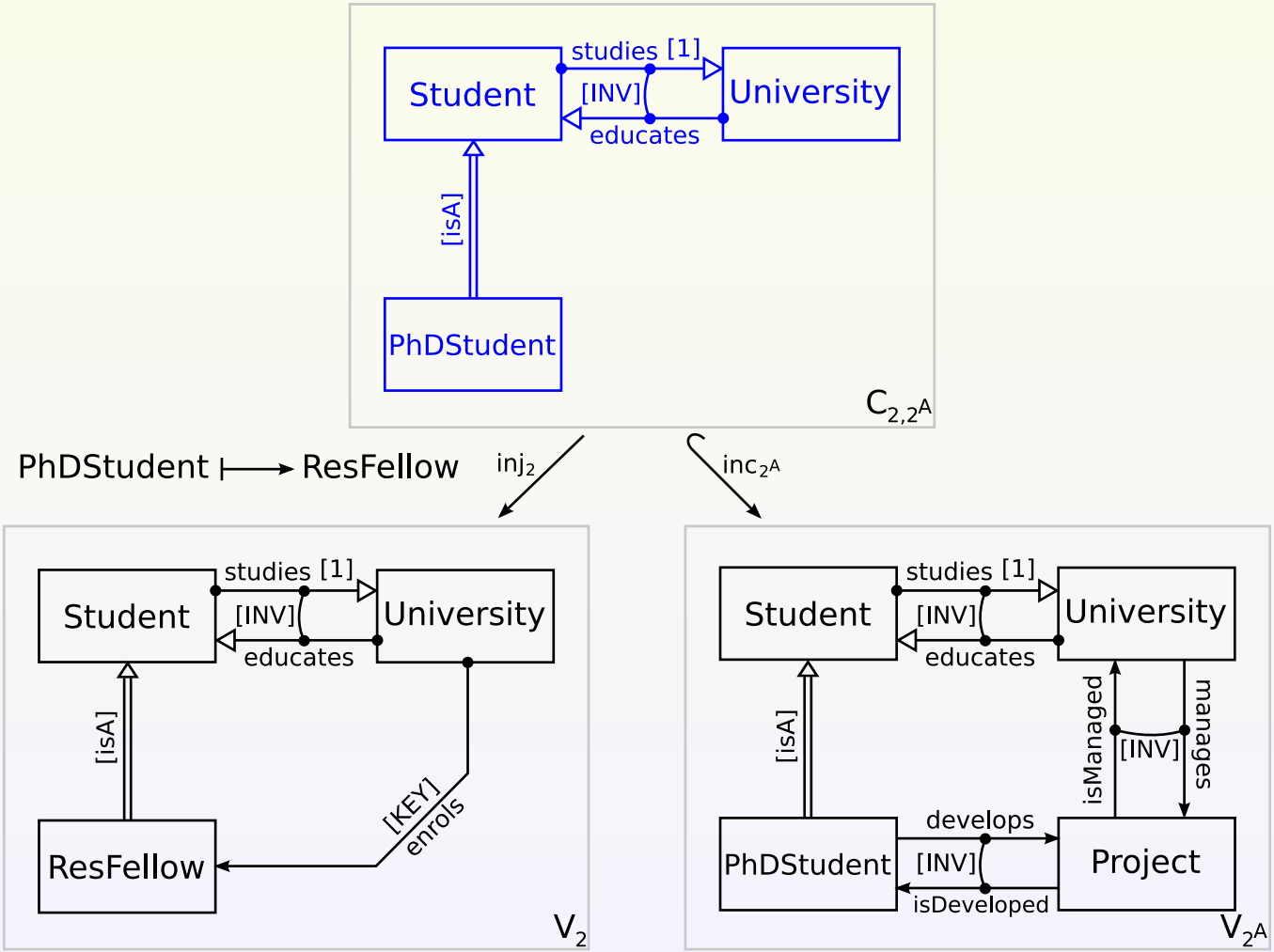
- We tackle the identification of mappings by specifying them explicitly in common models
- "Recording" which elements are kept unmodified addresses the problems of the soft- and hard-linking approaches

Definition

A model C together with the injective morphism inj and the inclusion morphism inc is a *common model* for V_i and V_j.

$$\begin{tikzcd} & C \arrow[dl, "inj"'] \arrow[dr, hook, "inc"] & \\ V_i & & V_j \end{tikzcd}$$

Example


Student
studies [1]
[INV]
educates
University
[isA]
PhDStudent
$C_{2,2^A}$
PhDStudent ⟼ ResFellow
inj_2
inc_{2^A}
Student
studies [1]
[INV]
educates
University
[isA]
[KEY]
enrols
ResFellow
V_2
Student
studies [1]
[INV]
educates
University
[isA]
isManaged
[INV]
manages
develops
PhDStudent
[INV]
isDeveloped
Project
V_{2^A}

Composition of Common Models

- Common model of non-subsequent models can be obtained by the composition of the "intermediate" common models

Definition

The *composition of commons* $C_{i,k}$ is a special pullback $(C_{i,k},\ inj_{C_{i,j}},\ inc_{C_{j,k}})$ of the diagram $C_{i,j} \xhookrightarrow{inc_{V_j}} V_j \xleftarrow{inj_{V_j}} C_{j,k}$ such that $inc_{C_{j,k}}$ is an inclusion

$$\begin{tikzcd}
 & & C_{i,k} \arrow[dl, hook, "inj_{C_{i,j}}"'] \arrow[dr, hook, "inc_{C_{j,k}}"] \arrow[dddll, bend right, "f"'] \arrow[dddrr, bend left, "g"] & & \\
 & C_{i,j} \arrow[dl, "inj_{V_i}"'] \arrow[dr, hook, "inc_{V_j}"] \arrow[r, phantom, "P.B."] & & C_{j,k} \arrow[dl, "inj_{V_j}"] \arrow[dr, hook, "inc_{V_k}"] & \\
V_i & & V_j & & V_k
\end{tikzcd}$$

Outline

Merge Model

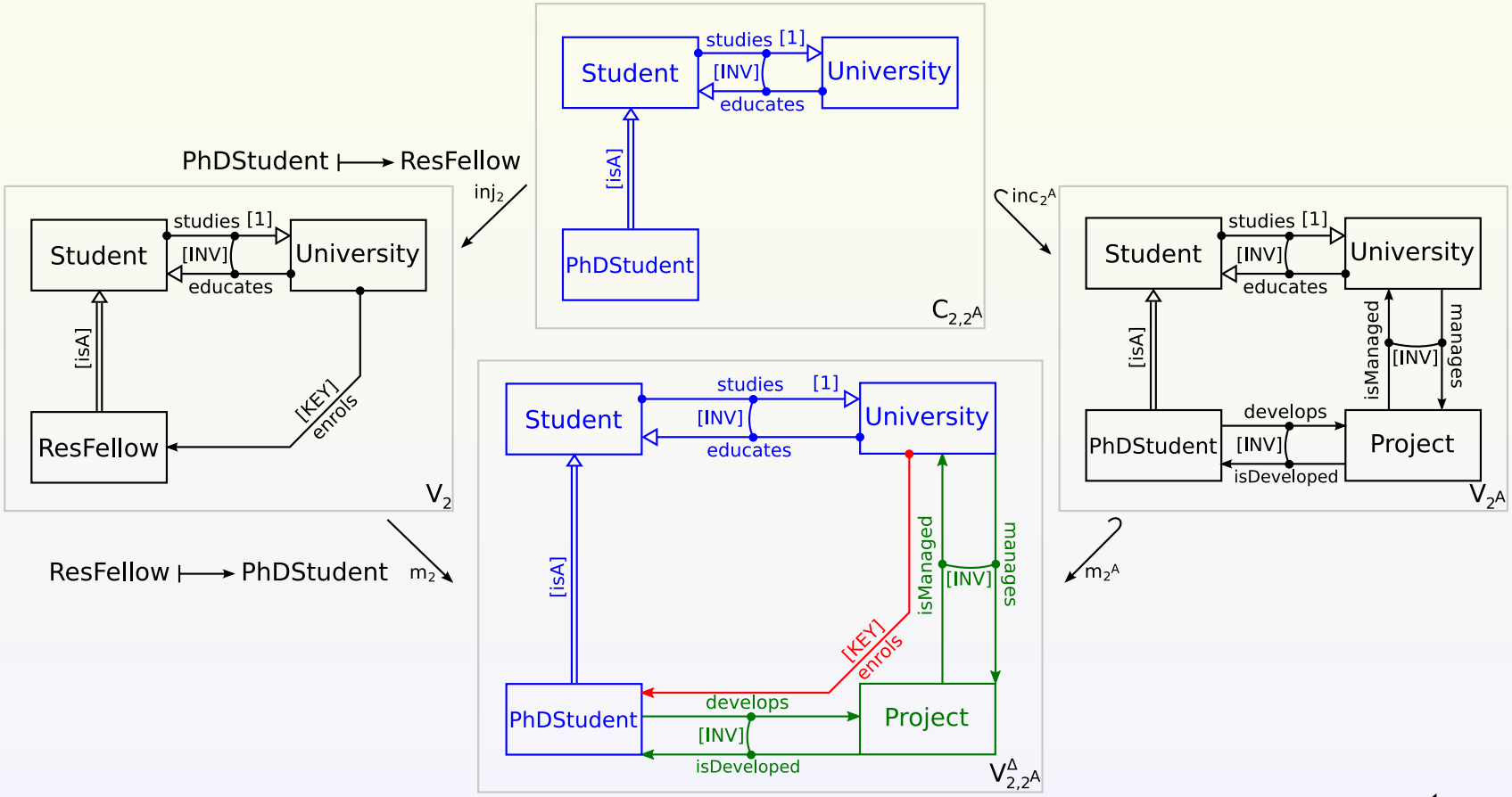
- The merge model must contain the information which is needed to distinguish which model elements come from which "source" model
  - This is exactly one of the properties of pushout

Definition

The *merge model* $V_{i,j}$ is the pushout $(V_{i,j},\ m_i,\ m_j)$ of the diagram $V_i \xleftarrow{inj_{V_i}} C_{i,j} \xhookrightarrow{inc_{V_j}} V_j$ such that m_j is an inclusion.

$$\begin{tikzcd}
 & C_{i,j} \arrow[dl, "inj_{V_i}"'] \arrow[dr, hook, "inc_{V_j}"] & \\
V_i \arrow[dr, "m_i"'] & P.O. & V_j \arrow[dl, hook, "m_j"] \\
 & V_{i,j} &
\end{tikzcd}$$

Example


PhDStudent ⟼ ResFellow
inj2
Student
studies [1]
[INV]
educates
University
[isA]
ResFellow
[KEY]
enrols
V2
ResFellow ⟼ PhDStudent
m2
Student
studies [1]
[INV]
educates
University
[isA]
PhDStudent
C2,2A
inc2A
Student
studies [1]
[INV]
educates
University
[isA]
isManaged
[INV]
manages
PhDStudent
develops
[INV]
isDeveloped
Project
V2A
m2A
Student
studies [1]
[INV]
educates
University
[isA]
isManaged
[INV]
manages
[KEY]
enrols
PhDStudent
develops
[INV]
isDeveloped
Project
V2,2A

Outline

Synchronisation and Commit

- A *synchronisation* is a procedure which generates a synchronised local copy V_{jU}, where V_j is the last version in the repository
  - conflicting changes are detected
- A *commit* is an operation which adds the model V_{jU} to the repository and labels it V_{j+1}

Synchronisation Procedure

$$
\begin{tikzcd}
 & C_{i,iU} \arrow[dl, hook, "inc_{V_{iU}}"'] \arrow[dr, "inj_{V_i}"] & & C_{i,j} \arrow[dl, "inj_{V_i}"'] \arrow[dr, hook, "inc_{V_j}"] & \\
V_{iU} \arrow[dr, hook] & & V_i \arrow[ll, Rightarrow, "e_{iU}"'] \arrow[rr, Rightarrow, "\cdots"] \arrow[dl, "mu_i"] \arrow[dr, "mr_i"'] & & V_j \arrow[dl, "mr_j"] \arrow[d, equal] \\
 & V_{i,iU} \arrow[rr, phantom, "P.O."] \arrow[dr, "mu_{iU,i}"'] & P.O. & V_{i,j} \arrow[dl, "mr_{i,j}"] & V_{j+1} \\
 & & V_{iU,j} \arrow[r, dashed] & V_{jU} \arrow[ur, Rightarrow, "c_{iU}"'] &
\end{tikzcd}
$$

Outline

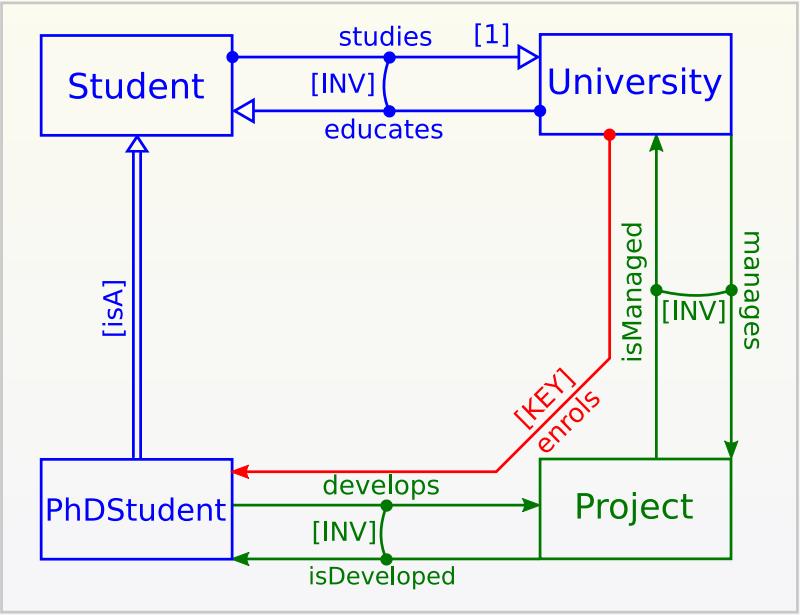
Difference

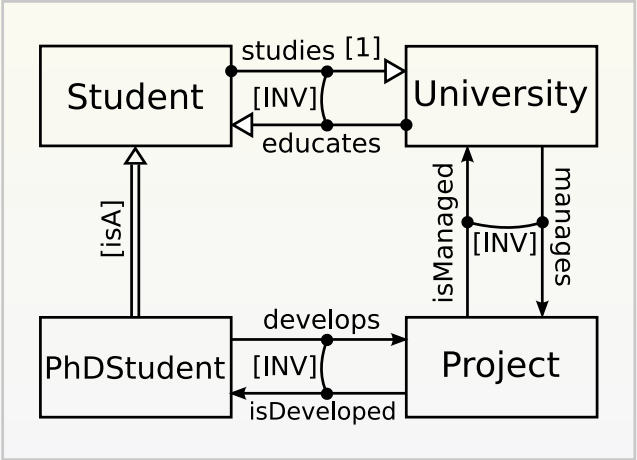
- Need to identify the differences between two models
- In a merge model $V_{i^U,j}$ we can distinguish common elements, V_{i^U}-elements and V_j-elements from each other
- We can identify which elements are *common*, *added*, *deleted*, *renamed* and *moved*
- We define a signature Σ_Δ
  - predicates: `[common]`, `[add]`, `[delete]`, `[rename]`, `[move]`, and `[conflict]`

Conflict

- The merge models will be decorated by predicates from the signature Σ_Δ
  - detection of conflicts in the decorated merge model $V_{i^U,j}$
  - reduction of $V_{i^U,j}$ to V_{j^U}
- Rules:
  - [common] $\wedge$ [common] $\rightarrow$ [common]
  - [delete] $\wedge$ [delete] $\rightarrow$ [delete]
  - [common] $\wedge$ [delete] $\rightarrow$ [delete]
  - [move] $\wedge$ [move] $\rightarrow$ [conflict]
  - [add] (arrow) $\wedge$ [delete] (node) $\rightarrow$ [conflict]
  - [rename] $\wedge$ [rename] $\rightarrow$ [conflict]

Example


Student
University
studies
[1]
[INV]
educates
[isA]
isManaged
[INV]
manages
[KEY]
enrols
PhDStudent
develops
[INV]
isDeveloped
Project


Student
studies [1]
[INV]
educates
University
[isA]
isManaged
[INV]
manages
PhDStudent
develops
[INV]
isDeveloped
Project

Outline

Summary

- The copy-modify-merge paradigm can be exploited for version control of graph-based structures
- Formalisation of the concepts of the copy-modify-merge paradigm in MDE by means of category-theoretical constructions
- Explicit specification of model elements' mappings in common models
- Diagrammatic language for the representation of model differences
- Rules for the detection of syntactic conflicts

Future Work

- Semantic conflicts
- Prototype implementation

Thank you!

Questions?