A Virtual Global Monorepo of Immutable Linked Data

Joeri Exelmans[1] Jakob Pietron[2] Alexander Raschke[2]
Hans Vangheluwe[1]

[1]University of Antwerp – Flanders Make, Belgium

[2]University of Ulm, Germany

MoM workshop @ MODELS
September 2024, Linz

Outline

What is Model Management?

Dealing with:

- Heterogeneous Models

What is Model Management?

Dealing with:

- Heterogeneous Models
- Inter-Dependencies

What is Model Management?

Dealing with:

- Heterogeneous Models
- Inter-Dependencies
- Information Overlaps

What is Model Management?

Dealing with:

- Heterogeneous Models
- Inter-Dependencies
- Information Overlaps
    - detecting and managing inconsistencies

What is Model Management?

Dealing with:

- Heterogeneous Models
- Inter-Dependencies
- Information Overlaps
    - detecting and managing inconsistencies
    - synchronization

What is Model Management?

Dealing with:

- Heterogeneous Models
- Inter-Dependencies
- Information Overlaps
  - detecting and managing inconsistencies
  - synchronization
- Evolution

What is Model Management?

Dealing with:

- Heterogeneous Models
- Inter-Dependencies
- Information Overlaps
    - detecting and managing inconsistencies
    - synchronization
- Evolution
    - . . . of models & languages

What is Model Management?

Dealing with:

- Heterogeneous Models
- Inter-Dependencies
- Information Overlaps
  - detecting and managing inconsistencies
  - synchronization
- Evolution
  - . . . of models & languages
  - concurrent updates, collaboration

What is Model Management?

Dealing with:

- Heterogeneous Models
- Inter-Dependencies
- Information Overlaps
  - detecting and managing inconsistencies
  - synchronization
- Evolution
  - . . . of models & languages
  - concurrent updates, collaboration

What is Model Management?

Dealing with:

- Heterogeneous Models
- Inter-Dependencies
- Information Overlaps
  - detecting and managing inconsistencies
  - synchronization
- Evolution
  - . . . of models & languages
  - concurrent updates, collaboration
- Access Control

“Research Question”

What would a MINIMAL CORE for model management look like?

And I also want . . .

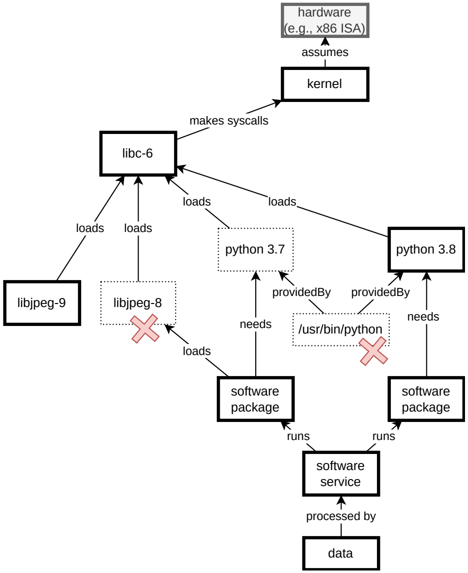
- Fully decentralized solution
- Access control without administration
- Eliminate software/model/data ‘rot’ (permanent replicability)

Outline

Software ‘rots’ . . .

- Software package developed 10 years ago, doesn’t work on my computer anymore . . .
- Why?

Software 'rots' . . .



Software ‘rots’ . . .

Entire issue can be reduced to **poor dependency management**

[1] e.g., npm, PyPI

Software 'rots' . . .

Entire issue can be reduced to **poor dependency management**

- Typical problems
  1. **implicit** dependencies
  2. **imprecise** dependencies
     (e.g., `/usr/bin/python` could be any Python version)
  3. **non-composable** dependencies
     (e.g., `/usr/bin/python`, naming collision)

[1] e.g., npm, PyPI

Software 'rots' . . .

Entire issue can be reduced to **poor dependency management**

- Typical problems
  1. **implicit** dependencies
  2. **imprecise** dependencies
     (e.g., `/usr/bin/python` could be any Python version)
  3. **non-composable** dependencies
     (e.g., `/usr/bin/python`, naming collision)

- Existing solutions

	(1)	(2)	(3)
Docker / VMs	sandboxing	flattening	
Lockfiles[1]		hash-based addressing	
Nix	sandboxing	hash-based addressing	hash-based addressing

[1] e.g., npm, PyPI

Centralization

Often-centralized features include

- state / storage
- access control
- issue tracking, pull requests, . . .

Centralization

Often-centralized features include

- state / storage
- access control
- issue tracking, pull requests, . . .

Centralization can be an **obstacle** to

- scalability
- cross-organization collaboration (next slide . . .)

Centralization

Often-centralized features include

- state / storage
- access control
- issue tracking, pull requests, . . .

Centralization can be an **obstacle** to

- scalability
- cross-organization collaboration (next slide . . .)

Decentralized systems can be deployed on a central server.
Distributing a centralized system can be very hard.

Access Control

Traditional approach = Access Control Lists

- users, groups, files / artifacts, access control rules

Access Control

Traditional approach = Access Control Lists

- users, groups, files / artifacts, access control rules
- rules manually specified $\Rightarrow$ administration overhead

Access Control

Traditional approach = Access Control Lists

- users, groups, files / artifacts, access control rules
- rules manually specified ⇒ administration overhead
- **simple** things are **hard**: give an external person access to an artifact?

Access Control

Traditional approach = Access Control Lists

- users, groups, files / artifacts, access control rules
- rules manually specified ⇒ administration overhead
- **simple** things are **hard**: give an external person access to an artifact?
  1. create user

Access Control

Traditional approach = Access Control Lists

- users, groups, files / artifacts, access control rules
- rules manually specified ⇒ administration overhead
- **simple** things are **hard**: give an external person access to an artifact?
    1. create user
    2. update AC rules

Access Control

Traditional approach = Access Control Lists

- users, groups, files / artifacts, access control rules
- rules manually specified ⇒ administration overhead
- **simple** things are **hard**: give an external person access to an artifact?
  1. create user
  2. update AC rules
  3. discover that the external person also needs access to dependencies of the artifact

Access Control

Traditional approach = Access Control Lists

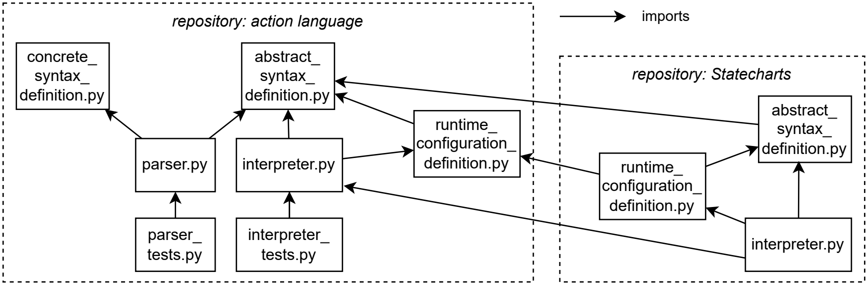
- users, groups, files / artifacts, access control rules
- rules manually specified ⇒ administration overhead
- **simple** things are **hard**: give an external person access to an artifact?
  1. create user
  2. update AC rules
  3. discover that the external person also needs access to dependencies of the artifact
  4. update AC rules

Access Control

Traditional approach = Access Control Lists

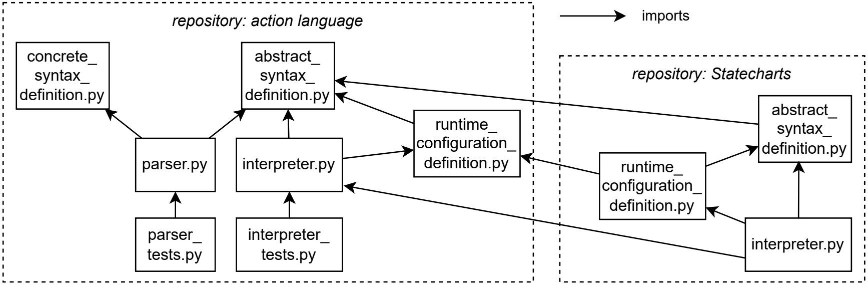
- users, groups, files / artifacts, access control rules
- rules manually specified $\Rightarrow$ administration overhead
- **simple** things are **hard**: give an external person access to an artifact?
  1. create user
  2. update AC rules
  3. discover that the external person also needs access to dependencies of the artifact
  4. update AC rules
  5. ...

Access Control


imports
repository: action language
concrete_ syntax_ definition.py
abstract_ syntax_ definition.py
parser.py
interpreter.py
runtime_ configuration_ definition.py
parser_ tests.py
interpreter_ tests.py
repository: Statecharts
abstract_ syntax_ definition.py
runtime_ configuration_ definition.py
interpreter.py

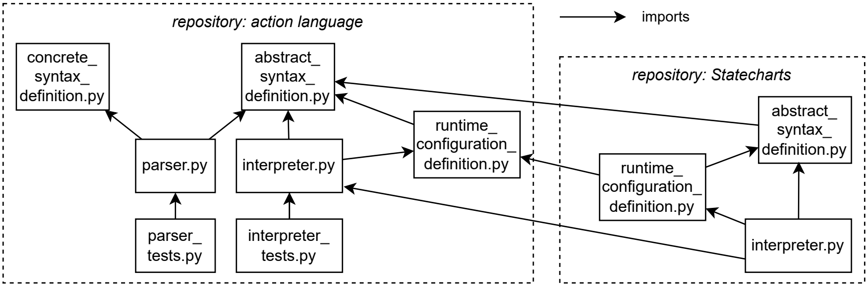
- access rights can be inconsistent with dependencies

Access Control


imports
repository: action language
concrete_ syntax_ definition.py
abstract_ syntax_ definition.py
parser.py
interpreter.py
runtime_ configuration_ definition.py
parser_ tests.py
interpreter_ tests.py
repository: Statecharts
abstract_ syntax_ definition.py
runtime_ configuration_ definition.py
interpreter.py

- access rights can be inconsistent with dependencies
- access control granted / revoked to
  - entire repository and its history (GitHub)
  - entire model

Access Control

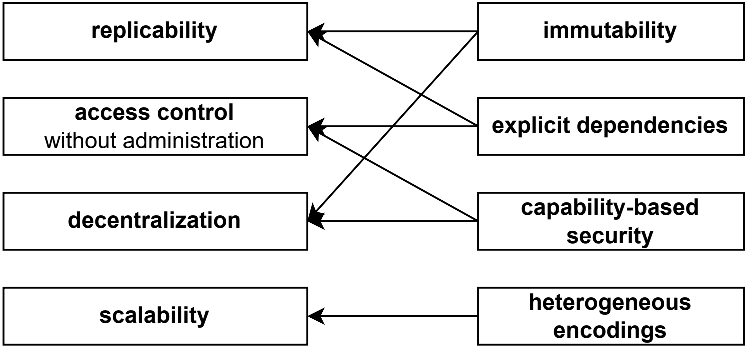

imports
repository: action language
concrete_ syntax_ definition.py
abstract_ syntax_ definition.py
parser.py
interpreter.py
runtime_ configuration_ definition.py
parser_ tests.py
interpreter_ tests.py
repository: Statecharts
abstract_ syntax_ definition.py
runtime_ configuration_ definition.py
interpreter.py

- access rights can be inconsistent with dependencies
- access control granted / revoked to
    - entire repository and its history (GitHub)
    - entire model

  which can be too much

Outline

Proposed Solution



Problem Space ← Solution Space

Immutability & Explicit Dependencies

All data is immutable

- **Bare minimum** requirement for replicability (software / data rot)
- Every object (=piece of data) gets permanent, unique ID
  - We propose hash-based IDs
- Update objects by *copy-on-write*

Immutability & Explicit Dependencies

All data is immutable

- **Bare minimum** requirement for replicability (software / data rot)
- Every object (=piece of data) gets permanent, unique ID
  - We propose hash-based IDs
- Update objects by *copy-on-write*

Explicit dependencies

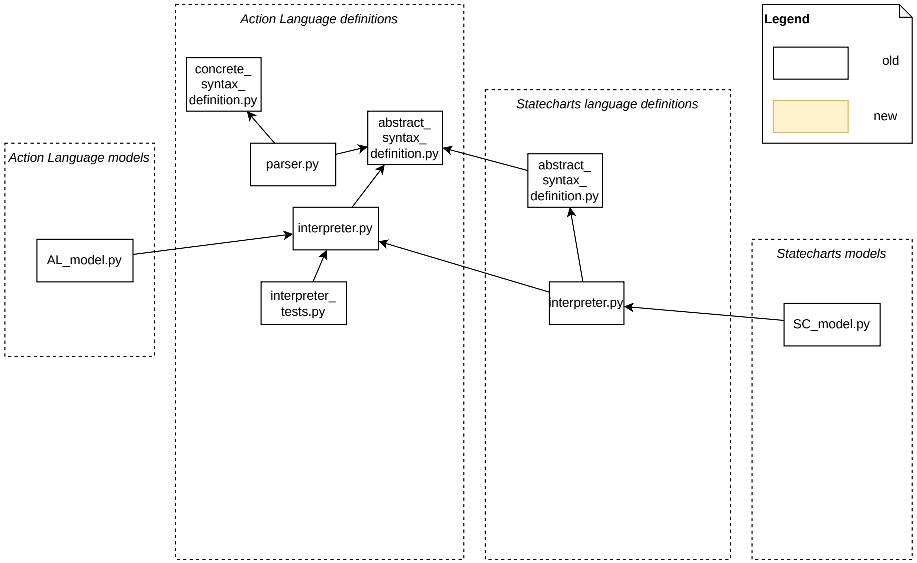
- An object must explicitly declare its dependencies (by referring to other hash-based IDs)
- An object's dependencies are 'part' of the object, and are immutable too

Examples of Dependencies

Examples

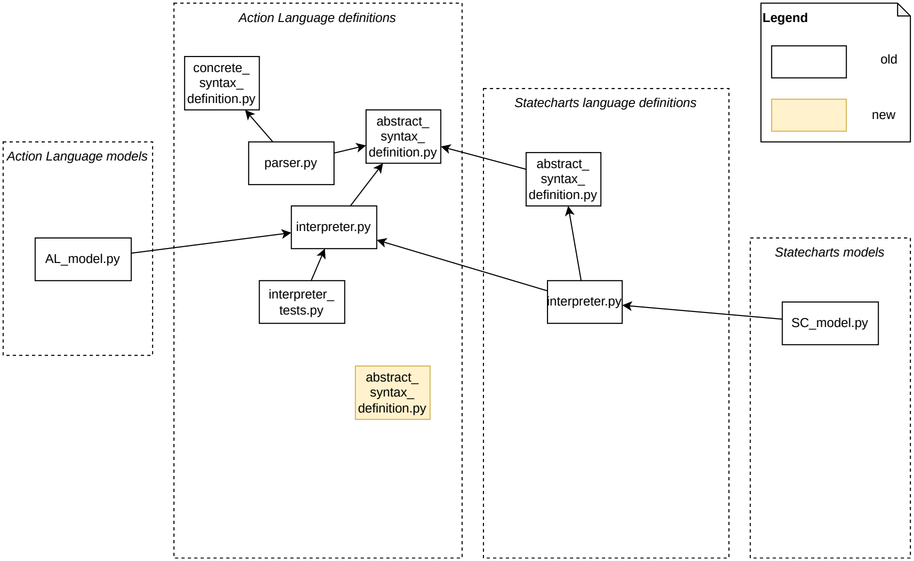
- A git-commit depends on the root of the snapshotted directory tree
- A directory depends on the items it contains
- A model depends on the model elements or sub-models it consists of
- A traceability link depends on the model elements it relates

Copy-on-write

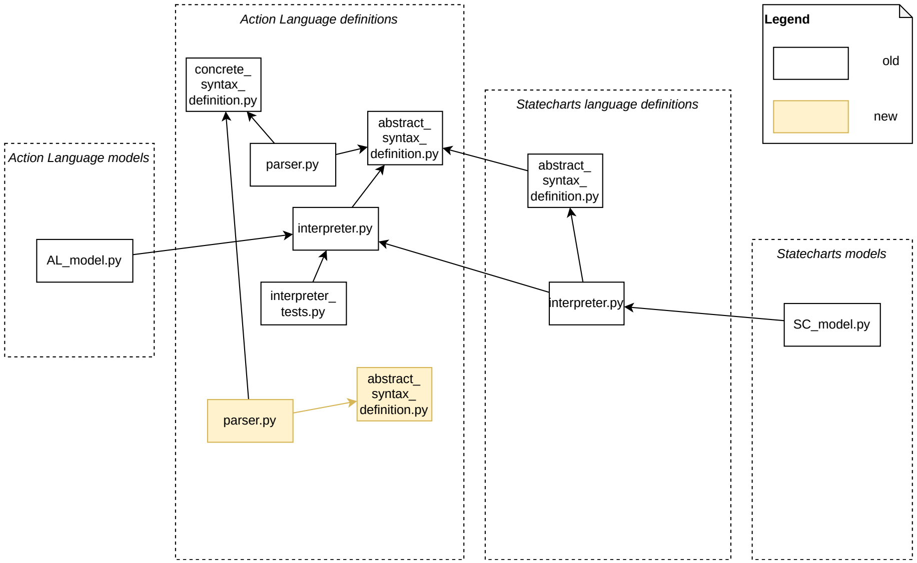


Action Language models
AL_model.py
Action Language definitions
concrete_ syntax_ definition.py
parser.py
abstract_ syntax_ definition.py
interpreter.py
interpreter_ tests.py
Statecharts language definitions
abstract_ syntax_ definition.py
interpreter.py
Statecharts models
SC_model.py
Legend
old
new

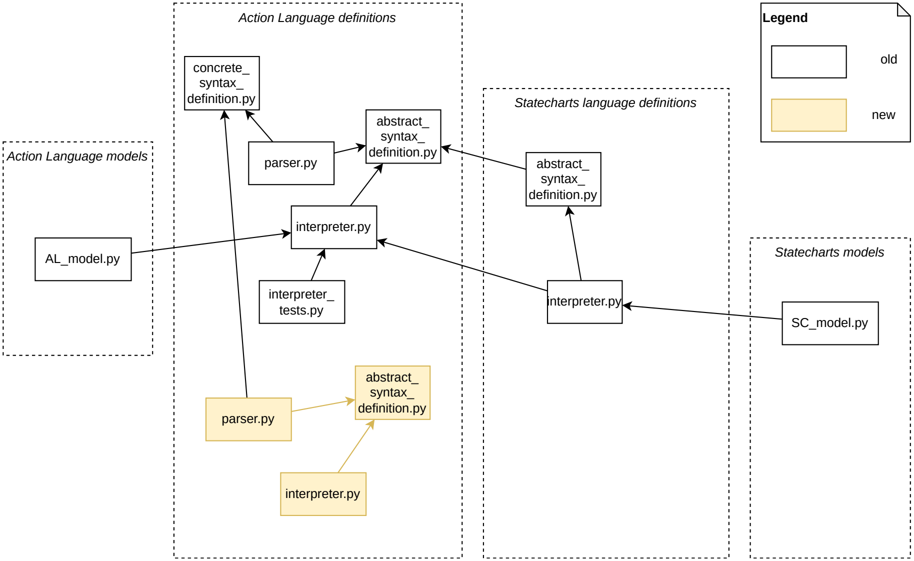
Copy-on-write


Action Language definitions
concrete_syntax_definition.py
parser.py
abstract_syntax_definition.py
interpreter.py
interpreter_tests.py
abstract_syntax_definition.py
Action Language models
AL_model.py
Statecharts language definitions
abstract_syntax_definition.py
interpreter.py
Statecharts models
SC_model.py
Legend
old
new

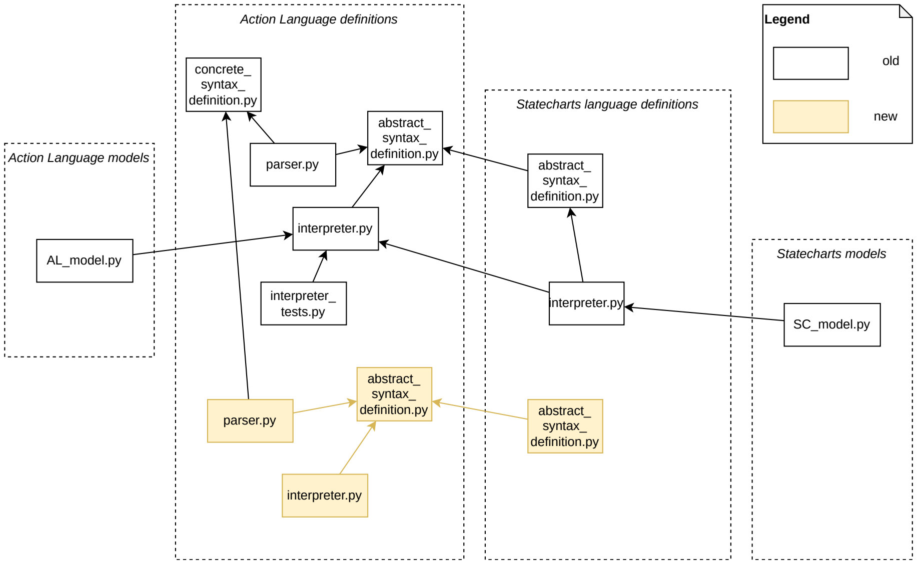
Copy-on-write


Action Language definitions
concrete_syntax_definition.py
parser.py
abstract_syntax_definition.py
interpreter.py
interpreter_tests.py
parser.py
abstract_syntax_definition.py
Action Language models
AL_model.py
Statecharts language definitions
abstract_syntax_definition.py
interpreter.py
Statecharts models
SC_model.py
Legend
old
new

Copy-on-write

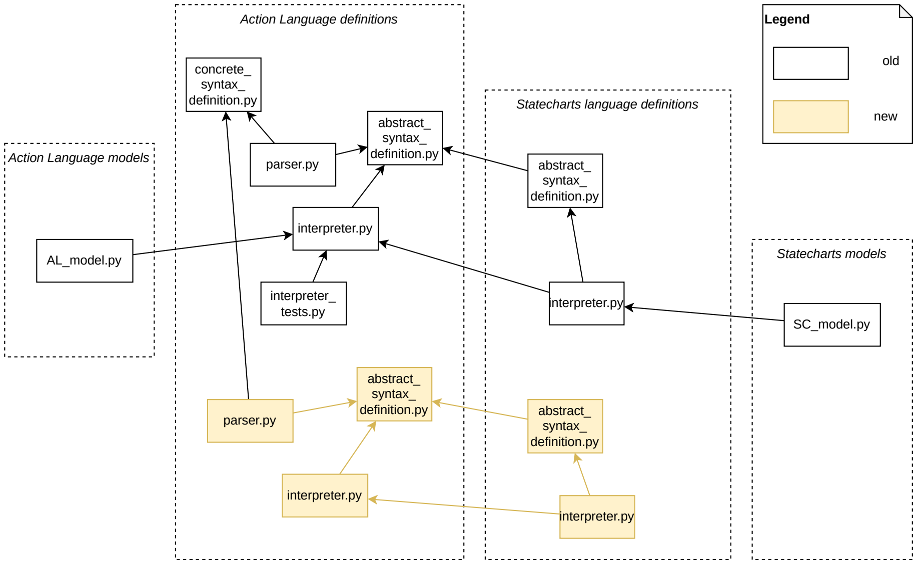

Action Language definitions
concrete_ syntax_ definition.py
abstract_ syntax_ definition.py
parser.py
interpreter.py
interpreter_ tests.py
parser.py
abstract_ syntax_ definition.py
interpreter.py
Action Language models
AL_model.py
Statecharts language definitions
abstract_ syntax_ definition.py
interpreter.py
Statecharts models
SC_model.py
Legend
old
new

Copy-on-write



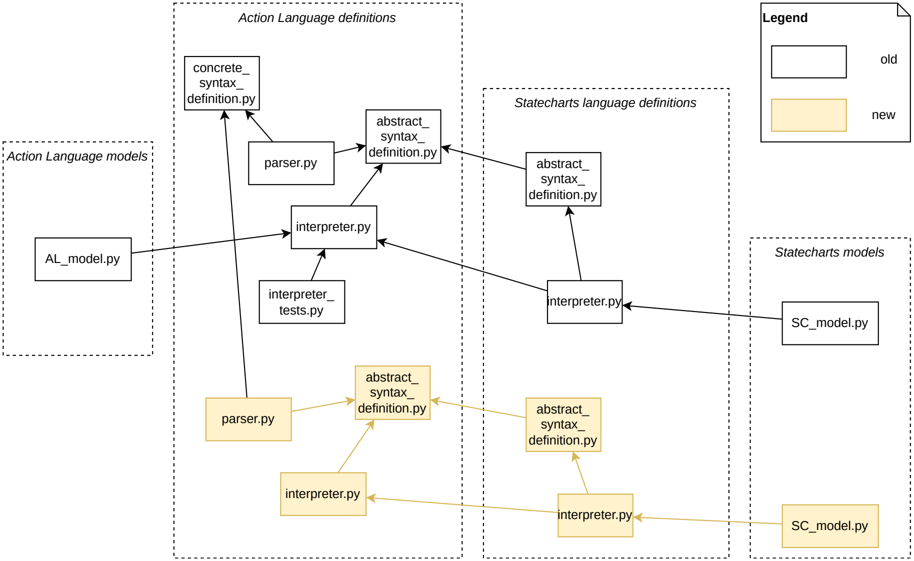
Action Language definitions
concrete_ syntax_ definition.py
parser.py
abstract_ syntax_ definition.py
interpreter.py
interpreter_ tests.py
parser.py
abstract_ syntax_ definition.py
interpreter.py
Action Language models
AL_model.py
Statecharts language definitions
abstract_ syntax_ definition.py
interpreter.py
abstract_ syntax_ definition.py
Statecharts models
SC_model.py
Legend
old
new

Copy-on-write



Action Language definitions
concrete_ syntax_ definition.py
parser.py
abstract_ syntax_ definition.py
interpreter.py
interpreter_ tests.py
parser.py
abstract_ syntax_ definition.py
interpreter.py
Action Language models
AL_model.py
Statecharts language definitions
abstract_ syntax_ definition.py
interpreter.py
abstract_ syntax_ definition.py
interpreter.py
Statecharts models
SC_model.py
Legend
old
new

Copy-on-write


Action Language definitions
concrete_ syntax_ definition.py
parser.py
abstract_ syntax_ definition.py
interpreter.py
interpreter_ tests.py
parser.py
abstract_ syntax_ definition.py
interpreter.py
Action Language models
AL_model.py
Statecharts language definitions
abstract_ syntax_ definition.py
interpreter.py
abstract_ syntax_ definition.py
interpreter.py
Statecharts models
SC_model.py
SC_model.py
Legend
old
new

Does "copy-on-write" scale?

[2] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.
[3] Ohad Rodeh. "B-trees, shadowing, and clones". In: *ACM Trans. Storage* 3.4 (2008).
[4] Sebastian Ullrich and Leonardo de Moura. "Counting immutable beans: reference counting optimized for purely functional programming". In: *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*. ACM, 2019, 3:1–3:12.

Does "copy-on-write" scale?

- Purely Functional Data Structures[2] have good time and space per-operation upper-bounds.
  - Sequences (Arrays, Linked Lists)
  - Sets
  - Dictionaries (Red-black trees, B-Trees[3])

[2] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.
[3] Ohad Rodeh. "B-trees, shadowing, and clones". In: *ACM Trans. Storage* 3.4 (2008).
[4] Sebastian Ullrich and Leonardo de Moura. "Counting immutable beans: reference counting optimized for purely functional programming". In: *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*. ACM, 2019, 3:1–3:12.

Does “copy-on-write” scale?

- Purely Functional Data Structures[2] have good time and space per-operation upper-bounds.
  - Sequences (Arrays, Linked Lists)
  - Sets
  - Dictionaries (Red-black trees, B-Trees[3])
- COW also used in enterprise filesystems: ZFS, BTRFS

[2] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[3] Ohad Rodeh. “B-trees, shadowing, and clones”. In: *ACM Trans. Storage* 3.4 (2008).

[4] Sebastian Ullrich and Leonardo de Moura. “Counting immutable beans: reference counting optimized for purely functional programming”. In: *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*. ACM, 2019, 3:1–3:12.

Does “copy-on-write” scale?

- Purely Functional Data Structures[2] have good time and space per-operation upper-bounds.
  - Sequences (Arrays, Linked Lists)
  - Sets
  - Dictionaries (Red-black trees, B-Trees[3])
- COW also used in enterprise filesystems: ZFS, BTRFS
- Statically detect when in-place overwrite can be done (Lean 4)[4]

[2] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[3] Ohad Rodeh. “B-trees, shadowing, and clones”. In: *ACM Trans. Storage* 3.4 (2008).

[4] Sebastian Ullrich and Leonardo de Moura. “Counting immutable beans: reference counting optimized for purely functional programming”. In: *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*. ACM, 2019, 3:1–3:12.

Does “copy-on-write” scale?

- Purely Functional Data Structures[2] have good time and space per-operation upper-bounds.
  - Sequences (Arrays, Linked Lists)
  - Sets
  - Dictionaries (Red-black trees, B-Trees[3])
- COW also used in enterprise filesystems: ZFS, BTRFS
- Statically detect when in-place overwrite can be done (Lean 4)[4]
- Zero-cost snapshots

[2] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[3] Ohad Rodeh. “B-trees, shadowing, and clones”. In: *ACM Trans. Storage* 3.4 (2008).

[4] Sebastian Ullrich and Leonardo de Moura. “Counting immutable beans: reference counting optimized for purely functional programming”. In: *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*. ACM, 2019, 3:1–3:12.

Does "copy-on-write" scale?

- Purely Functional Data Structures[2] have good time and space per-operation upper-bounds.
    - Sequences (Arrays, Linked Lists)
    - Sets
    - Dictionaries (Red-black trees, B-Trees[3])
- COW also used in enterprise filesystems: ZFS, BTRFS
- Statically detect when in-place overwrite can be done (Lean 4)[4]
- Zero-cost snapshots
- The alternative, "write-in-place" can only support snapshots with expensive deep-copy.

[2] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[3] Ohad Rodeh. "B-trees, shadowing, and clones". In: *ACM Trans. Storage* 3.4 (2008).

[4] Sebastian Ullrich and Leonardo de Moura. "Counting immutable beans: reference counting optimized for purely functional programming". In: *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*. ACM, 2019, 3:1–3:12.

Access Control – Capability-Based Security

We apply the principles of Capability-Based Security

Access Control – Capability-Based Security

We apply the principles of Capability-Based Security

- No need to restrict **write** access (all data is immutable)

Access Control – Capability-Based Security

We apply the principles of Capability-Based Security

- No need to restrict **write** access (all data is immutable)
- **Read** access is restricted:
    - You can only see data you created, and data that was explicitly shared with you
    - Data is shared by sharing its hash-based ID (‘share-by-link’)
    - Having read-access to a piece of data, implies having read-access to its dependencies (but not vice versa)

Decentralized Data Store

- Minimal interface:

```
 get(hash) -> object
put(object) -> hash
```

Decentralized Data Store

- Minimal interface:

```
 get(hash) -> object
put(object) -> hash
```

- Objects can exist in multiple stores (local, remote, cloud, . . .)

Decentralized Data Store

- Minimal interface:

```
 get(hash) -> object
put(object) -> hash
```

- Objects can exist in multiple stores (local, remote, cloud, . . .)
- Synchronization between stores is trivial:
  - a store either has an object or not

Decentralized Data Store

- Minimal interface:
  ```
   get(hash) -> object
  put(object) -> hash
  ```
- Objects can exist in multiple stores (local, remote, cloud, . . .)
- Synchronization between stores is trivial:
  - a store either has an object or not
- Objects & dependencies forms a DAG (by construction)
  - can use *reference counting* AKA *poor man's garbage collection* to clean up unreachable data

Heterogeneous Encodings

Heterogeneous Encodings

- Don't want to transform all data to some uniform encoding (e.g., graph, XMI)

Heterogeneous Encodings

- Don't want to transform all data to some uniform encoding (e.g., graph, XMI)
- Instead keep data in its original encoding

Heterogeneous Encodings

- Don't want to transform all data to some uniform encoding (e.g., graph, XMI)
- Instead keep data in its original encoding
- All encodings provided by 'plugins' / *adapters*

Heterogeneous Encodings

- Don't want to transform all data to some uniform encoding (e.g., graph, XMI)
- Instead keep data in its original encoding
- All encodings provided by 'plugins' / *adapters*
- Adapters implement **interfaces**:
  - mandatory: get dependencies, (de-)serialization
  - optional:
    - models, directories, . . . : expose inner structure
    - executable models: initialize, execute, . . .
    - versions (of anything): merge
    - numbers: arithmetic

Heterogeneous Encodings

- Don't want to transform all data to some uniform encoding (e.g., graph, XMI)
- Instead keep data in its original encoding
- All encodings provided by 'plugins' / *adapters*
- Adapters implement **interfaces**:
  - mandatory: get dependencies, (de-)serialization
  - optional:
    - models, directories, . . . : expose inner structure
    - executable models: initialize, execute, . . .
    - versions (of anything): merge
    - numbers: arithmetic
- No 'builtin' / 'native' encodings!

Outline

Versioning

Versioning

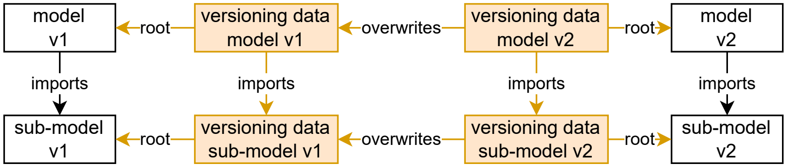
- Versioning information is **just data**

Versioning

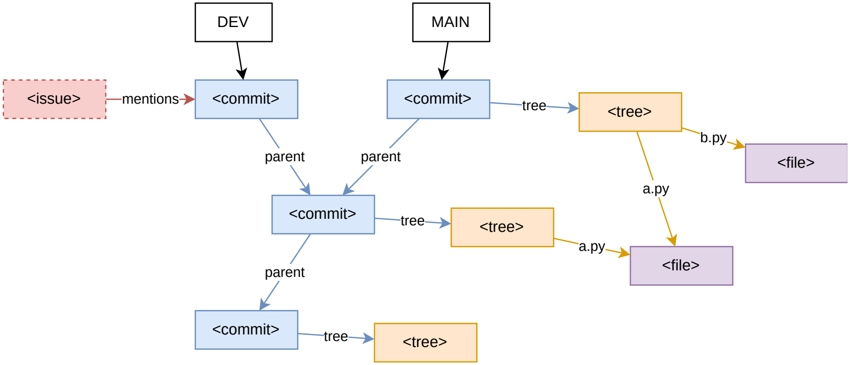
- Versioning information is **just data**
- No need to enforce one particular approach to versioning

Versioning

- Versioning information is **just data**
- No need to enforce one particular approach to versioning
- Currently experimenting with hierarchical versioning approach that records all causes of computed values



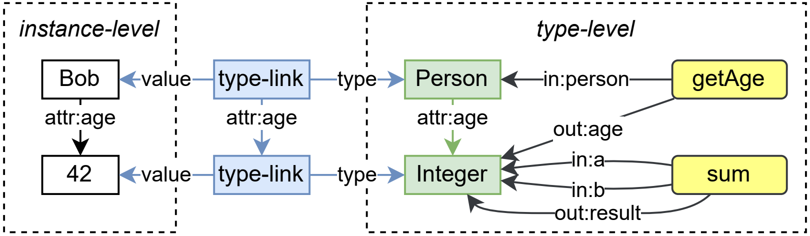
Versioning – Git integration



A git repo is already a store of immutable objects.
Easy to integrate!

Typing

- Typing information is **just data**



- type-level data is FTG

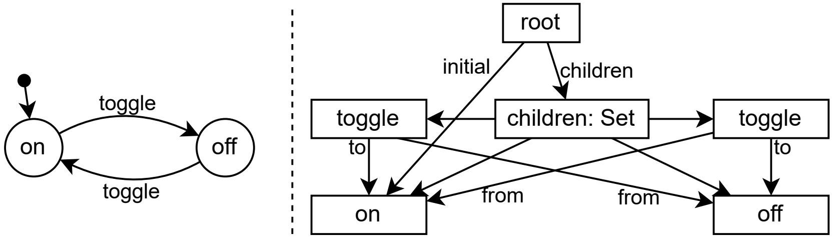
Outline

Challenges

- Scalability (cost of copy-on-write – already discussed)
- Inability to revoke read-access
- Inability to encode cyclic references (next slide . . .)

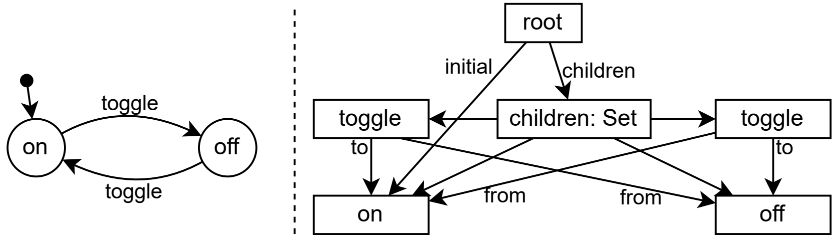
Breaking up cycles

We follow WebGME's approach:



Breaking up cycles

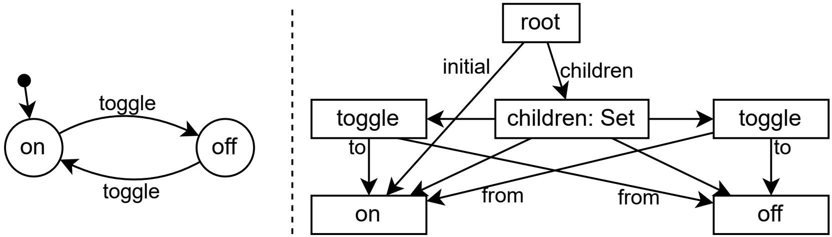
We follow WebGME's approach:



- Costly to find a state's outgoing transitions (needed to simulate)

Breaking up cycles

We follow WebGME's approach:



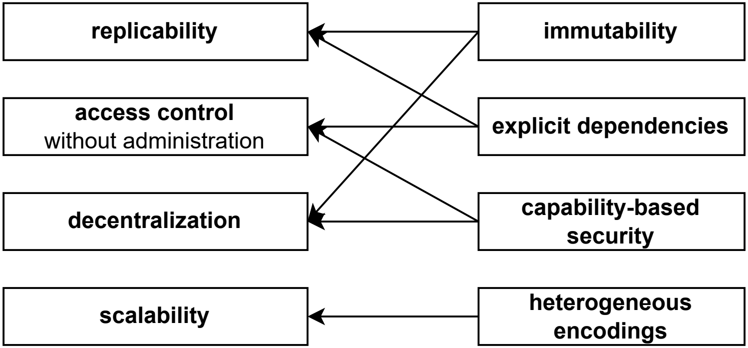
- Costly to find a state's outgoing transitions (needed to simulate)
- BUT we can *memoize* this information (due to immutability)

Conclusions

We presented a **decentralized data model** for model management.

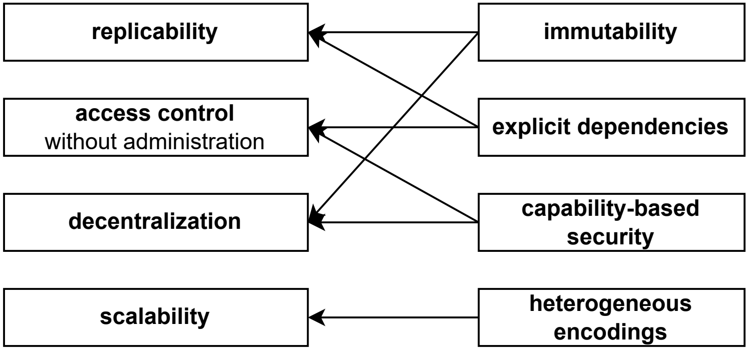
Conclusions

We presented a **decentralized data model** for model management.



Conclusions

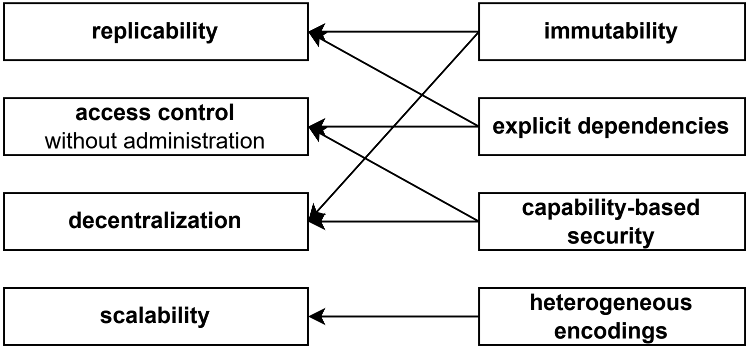
We presented a **decentralized data model** for model management.



- **Access control** is **fundamental**

Conclusions

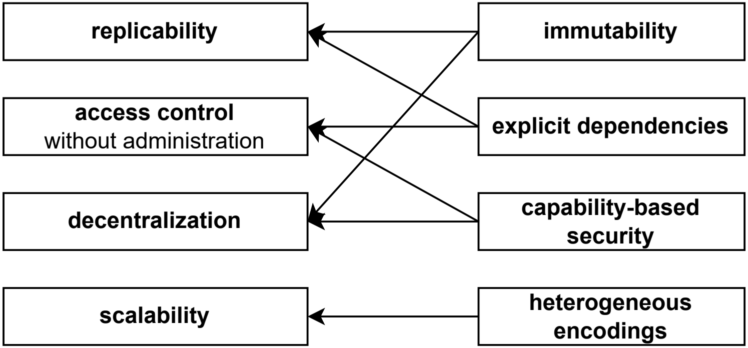
We presented a **decentralized data model** for model management.



- **Access control** is **fundamental**
- **Immutability** seems expensive, but **pays off** (in our opinion)

Conclusions

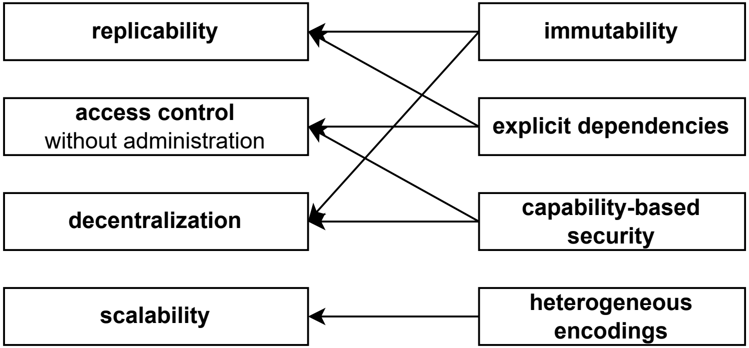
We presented a **decentralized data model** for model management.



- **Access control** is **fundamental**
- **Immutability** seems expensive, but **pays off** (in our opinion)
- Surprise: **Versioning** is **not** a **fundamental** feature,

Conclusions

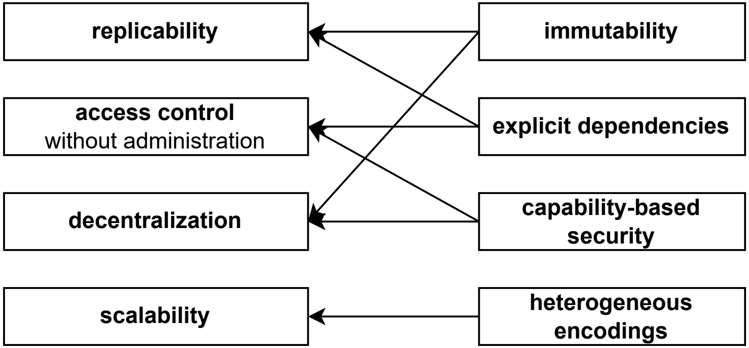
We presented a **decentralized data model** for model management.



- **Access control** is **fundamental**
- **Immutability** seems expensive, but **pays off** (in our opinion)
- Surprise: **Versioning** is **not** a **fundamental** feature, but **immutability is**

Conclusions

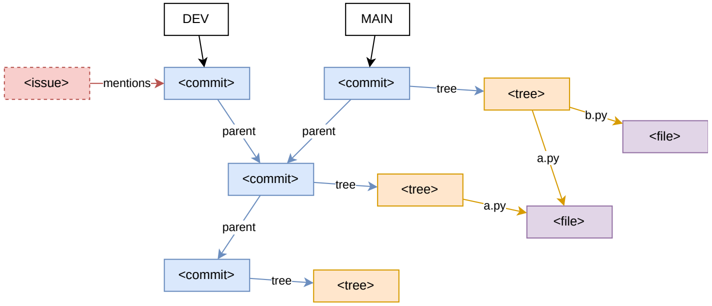
We presented a **decentralized data model** for model management.



- **Access control** is **fundamental**
- **Immutability** seems expensive, but **pays off** (in our opinion)
- Surprise: **Versioning** is **not** a **fundamental** feature, but **immutability is**
- Easy to integrate immutable data (e.g., git)

Backup slide

- Write-in-place + storing (immutable) deltas has one big shortcoming: The deltas are 'special' (separate from the data being versioned)
- Cannot treat versions / deltas as data


DEV
MAIN
<issue>
mentions
<commit>
<commit>
tree
<tree>
b.py
<file>
parent
parent
a.py
<commit>
tree
<tree>
a.py
<file>
parent
<commit>
tree
<tree>