Scalable Armies of Model Clones through Data Sharing

MODELS 2014 Valencia

Erwan Bousse[1] Benoit Combemale[2] Benoit Baudry[2]

[1]University of Rennes 1 (IRISA), France

[2]Inria, France

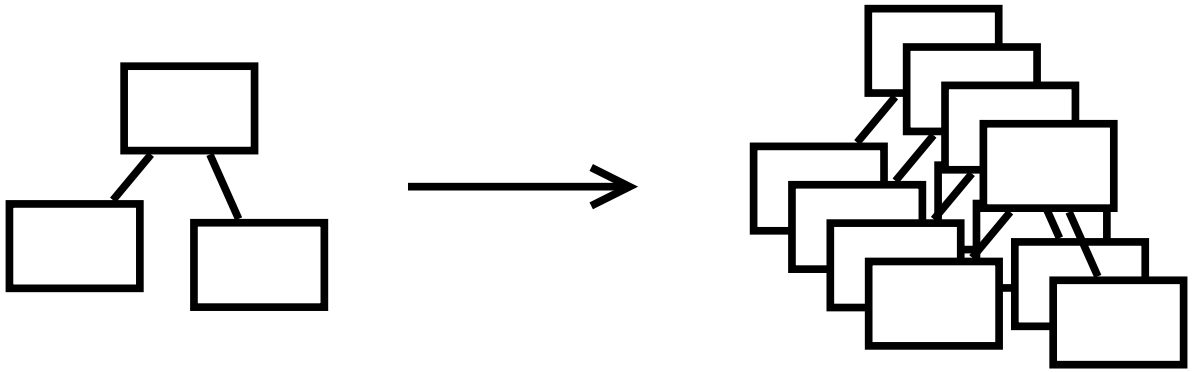
November 24, 2014

Outline

Plan

Context and motivation

New MDE activities which rely on the production of **large quantities of models and variations of a set of models**, that can be obtained through **model cloning**.



Definition: clone

A *clone* is a model that is, when created, is identical to an existing model. Both models conform to the same metamodel and are independent from one to another.

Example of large quantities of clones

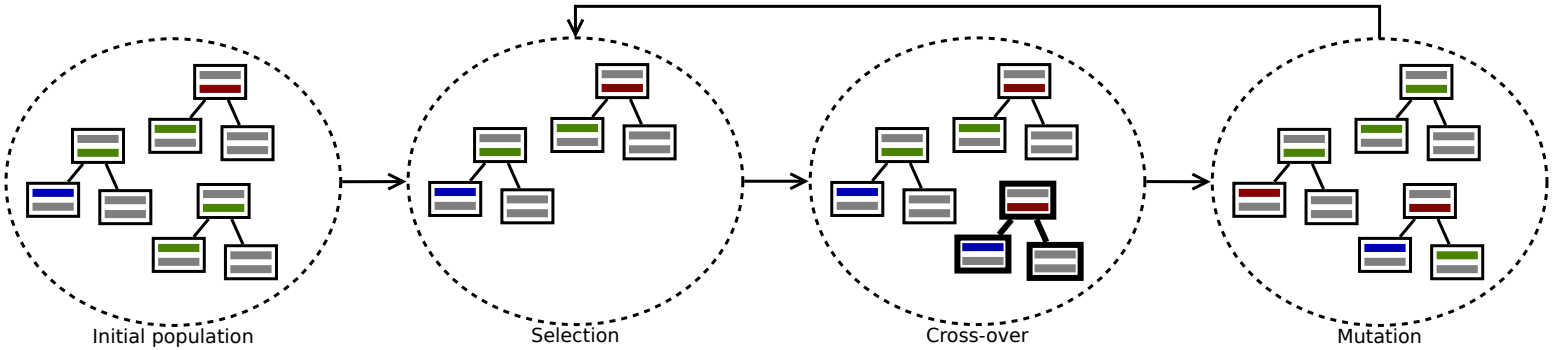
Evolutionnary computation:

Execution trace:

Design space exploration:

Example of large quantities of clones

Evolutionnary computation:

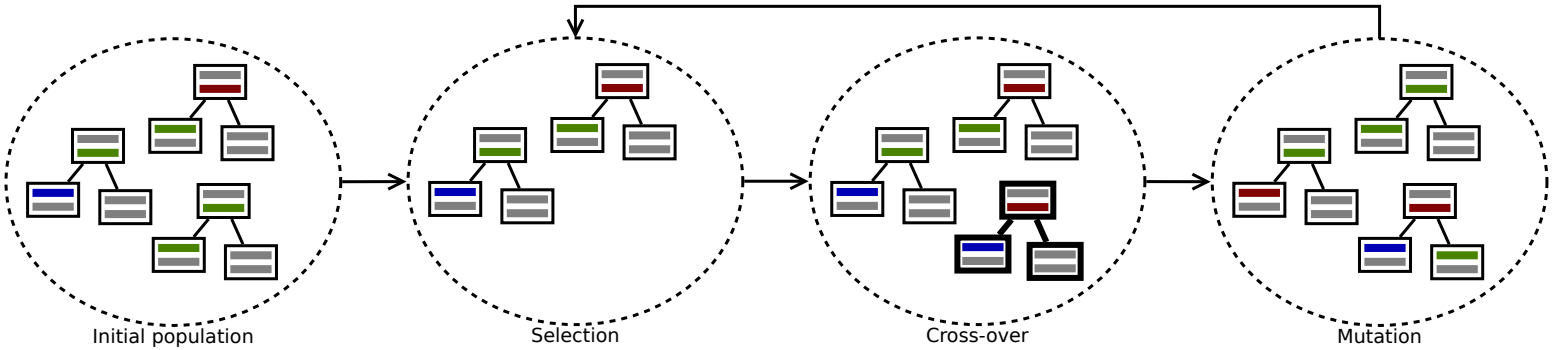


Execution trace:

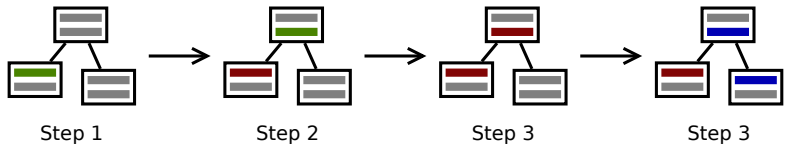
Design space exploration:

Example of large quantities of clones

Evolutionnary computation:



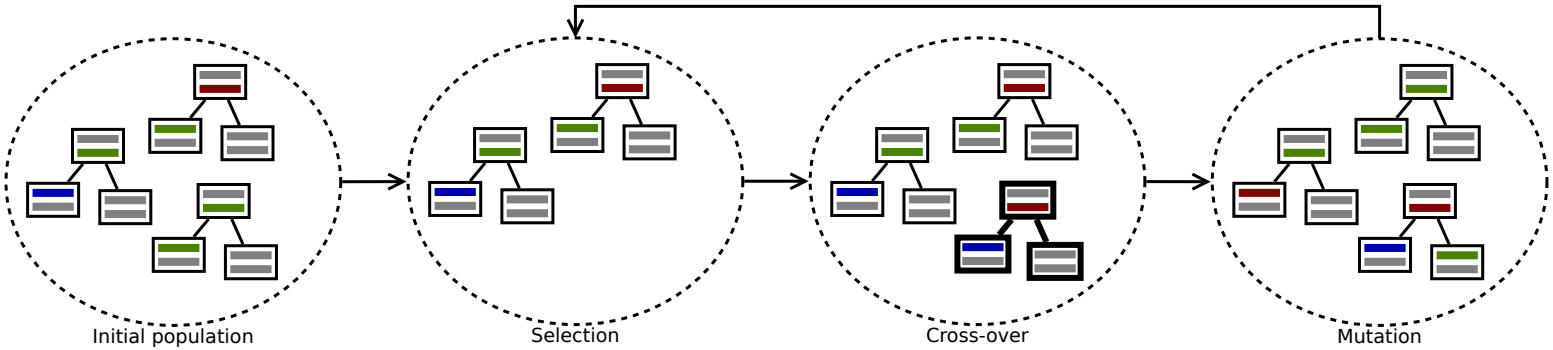
Execution trace:



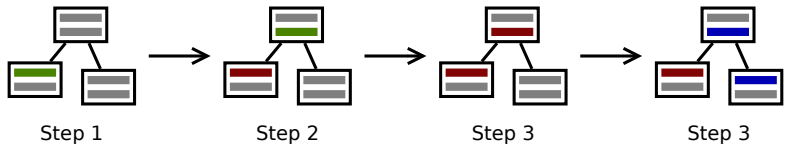
Design space exploration:

Example of large quantities of clones

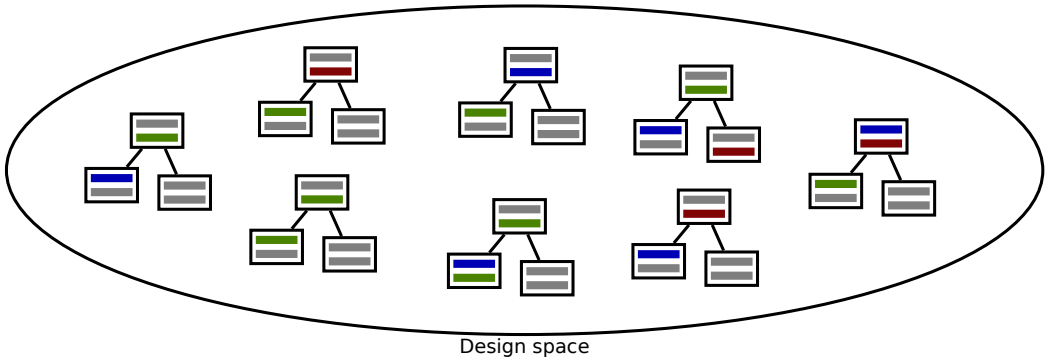
Evolutionnary computation:



Execution trace:



Design space exploration:



Problem: efficient model cloning

- Need for the ability to **clone a model**
- Already possible using the most convenient cloning implementation: **deep cloning** (see *EcoreUtil.Copier* class)
- *deep cloning* $\equiv$ duplicating the model in memory

Problem: *deep cloning* has very poor memory performances

Problem: efficient model cloning

- Need for the ability to **clone a model**
- Already possible using the most convenient cloning implementation: **deep cloning** (see *EcoreUtil.Copier* class)
- *deep cloning* $\equiv$ duplicating the model in memory

Problem: *deep cloning* has very poor memory performances

Plan

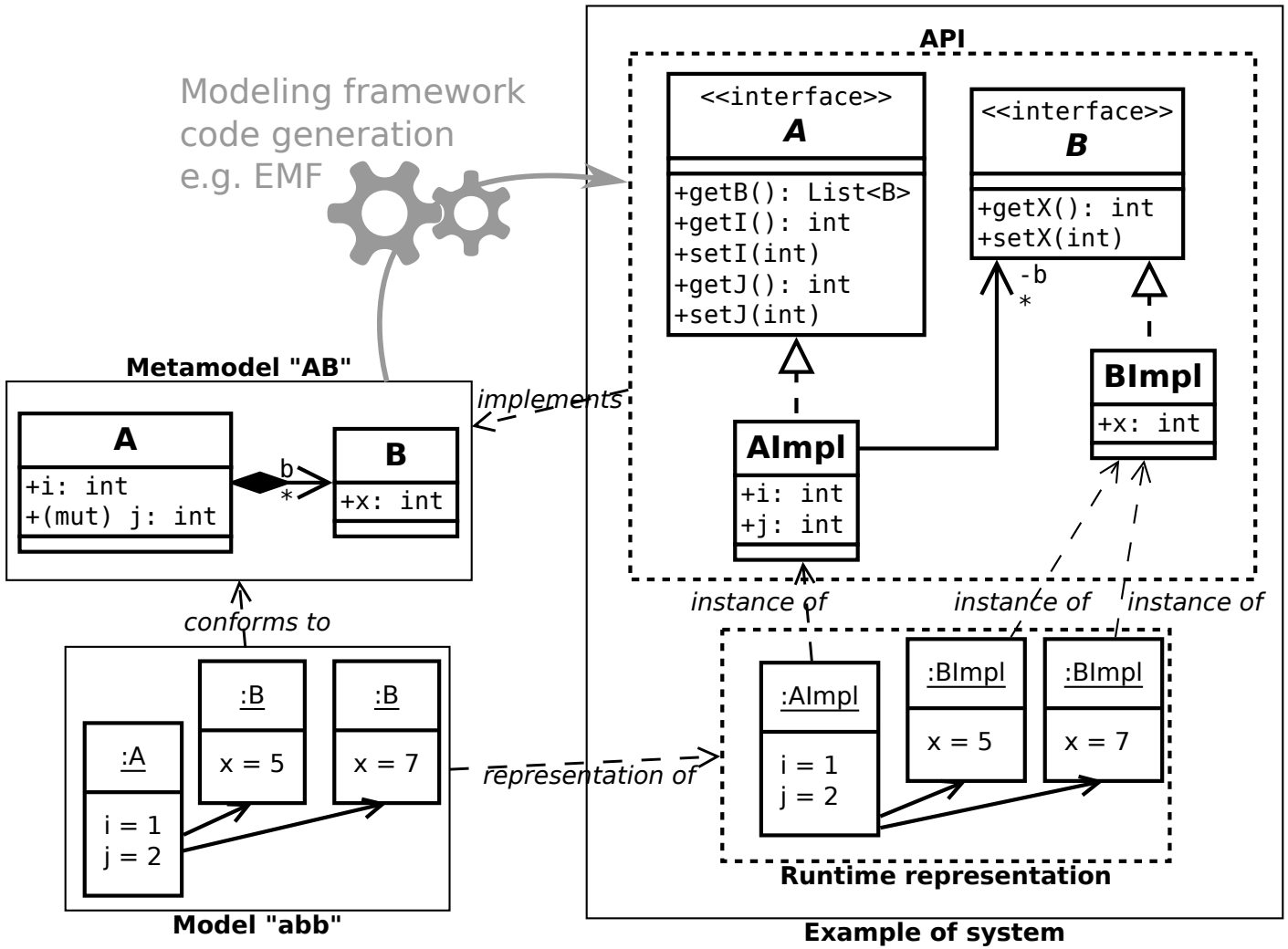
Manipulating metamodels and models at runtime

- **Execution environments** are necessary to make concrete use of metamodels and models
- One of the most popular: Eclipse Modeling Framework (EMF)
- EMF generates Java interfaces and classes that implement a metamodel, providing mechanisms to create **runtime representations** of models that conform to the metamodel

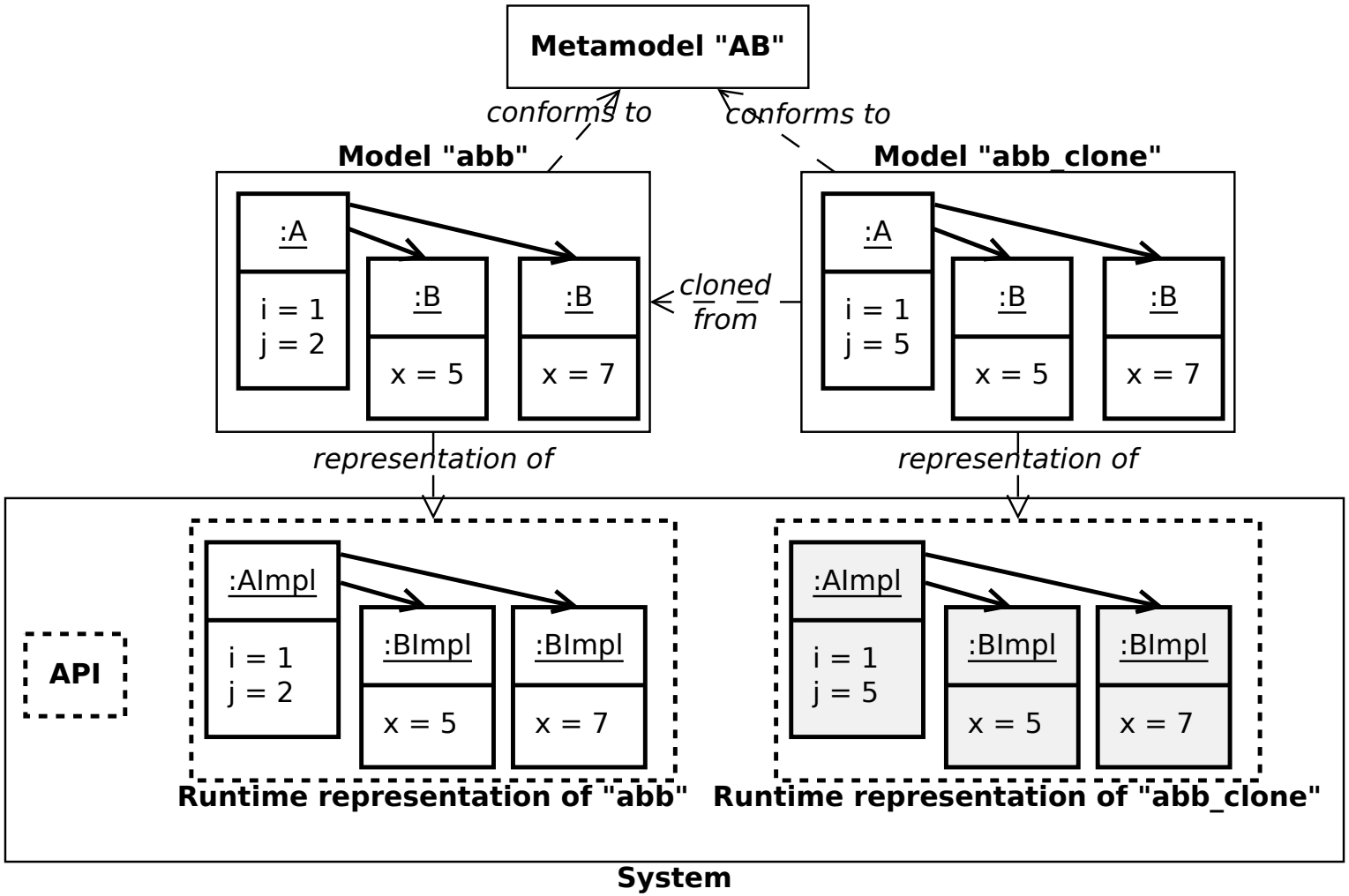
Definition: runtime representation

The *runtime representation* of a model is **the set of runtime data that is sufficient to reflect the model data structure**. It must be manipulated through an interface that is consistent with the corresponding metamodel.

Example: a metamodel, a model, and runtime counterparts



Example: a model clone obtained using *deep cloning*



Plan

Requirements for cloning operators

Req #1 scalability.

Runtime representations of model clones must scale in memory.

Req #2 manipulation performance.

It is necessary to manipulate the clones as efficiently as any model.

Req #3 model interface.

The clones and the original model must be manipulated through the same interface.

Req #4 reflective layer

Support model manipulation through a reflective layer.

Requirements for cloning operators

Req #1 scalability.

Runtime representations of model clones must scale in memory.

Req #2 manipulation performance.

It is necessary to manipulate the clones as efficiently as any model.

Req #3 model interface.

The clones and the original model must be manipulated through the same interface.

Req #4 reflective layer

Support model manipulation through a reflective layer.

Requirements for cloning operators

Req #1 scalability.

Runtime representations of model clones must scale in memory.

Req #2 manipulation performance.

It is necessary to manipulate the clones as efficiently as any model.

Req #3 model interface.

The clones and the original model must be manipulated through the same interface.

Req #4 reflective layer

Support model manipulation through a reflective layer.

Requirements for cloning operators

Req #1 scalability.

Runtime representations of model clones must scale in memory.

Req #2 manipulation performance.

It is necessary to manipulate the clones as efficiently as any model.

Req #3 model interface.

The clones and the original model must be manipulated through the same interface.

Req #4 reflective layer

Support model manipulation through a reflective layer.

Requirements for cloning operators

Req #1 scalability.

Runtime representations of model clones must scale in memory.

Req #2 manipulation performance.

It is necessary to manipulate the clones as efficiently as any model.

Req #3 model interface.

The clones and the original model must be manipulated through the same interface.

Req #4 reflective layer

Support model manipulation through a reflective layer.

Data sharing: existing approaches

Idea for Req #1: considering that only a subset of a model changes during its lifecycle, **avoid data redundancy among clones**

Dynamically: copy-on-write (aka lazy copy)

Create virtual copies, and create real copies on write accesses.

- either using a specific API/entry-point, **breaks Req#3**
- or in a transparent way, but managing consistency depends on the implementation language (e.g. Java is pass-by-value)
- Copies are done during manipulations, **may break Req#2**

Statically: flyweight design pattern

Objects are designed to be used in multiple contexts.

- Requires the passing the extrinsic state (ie the mutable part) of the object as a parameter, for all its operations, **breaks Req #3**

Data sharing: existing approaches

Idea for Req #1: considering that only a subset of a model changes during its lifecycle, **avoid data redundancy among clones**

Dynamically: copy-on-write (aka lazy copy)

Create virtual copies, and create real copies on write accesses.

- either using a specific API/entry-point, **breaks Req#3**
- or in a transparent way, but managing consistency depends on the implementation language (e.g. Java is pass-by-value)
- Copies are done during manipulations, **may break Req#2**

Statically: flyweight design pattern

Objects are designed to be used in multiple contexts.

- Requires the passing the extrinsic state (ie the mutable part) of the object as a parameter, for all its operations, **breaks Req #3**

Data sharing: existing approaches

Idea for Req #1: considering that only a subset of a model changes during its lifecycle, **avoid data redundancy among clones**

Dynamically: copy-on-write (aka lazy copy)

Create virtual copies, and create real copies on write accesses.

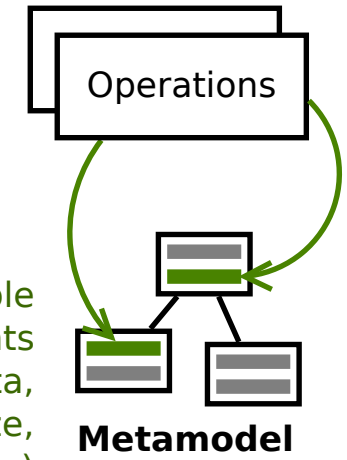
- either using a specific API/entry-point, **breaks Req#3**
- or in a transparent way, but managing consistency depends on the implementation language (e.g. Java is pass-by-value)
- Copies are done during manipulations, **may break Req#2**

Statically: flyweight design pattern

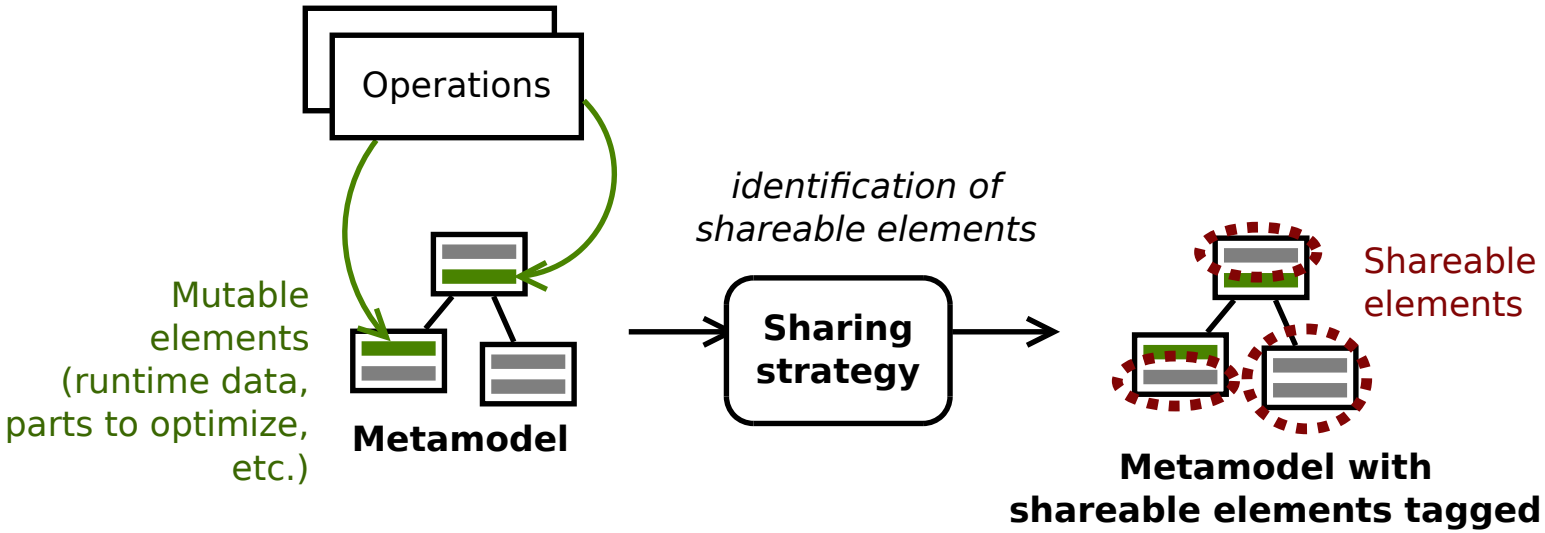
Objects are designed to be used in multiple contexts.

- Requires the passing the extrinsic state (ie the mutable part) of the object as a parameter, for all its operations, **breaks Req #3**

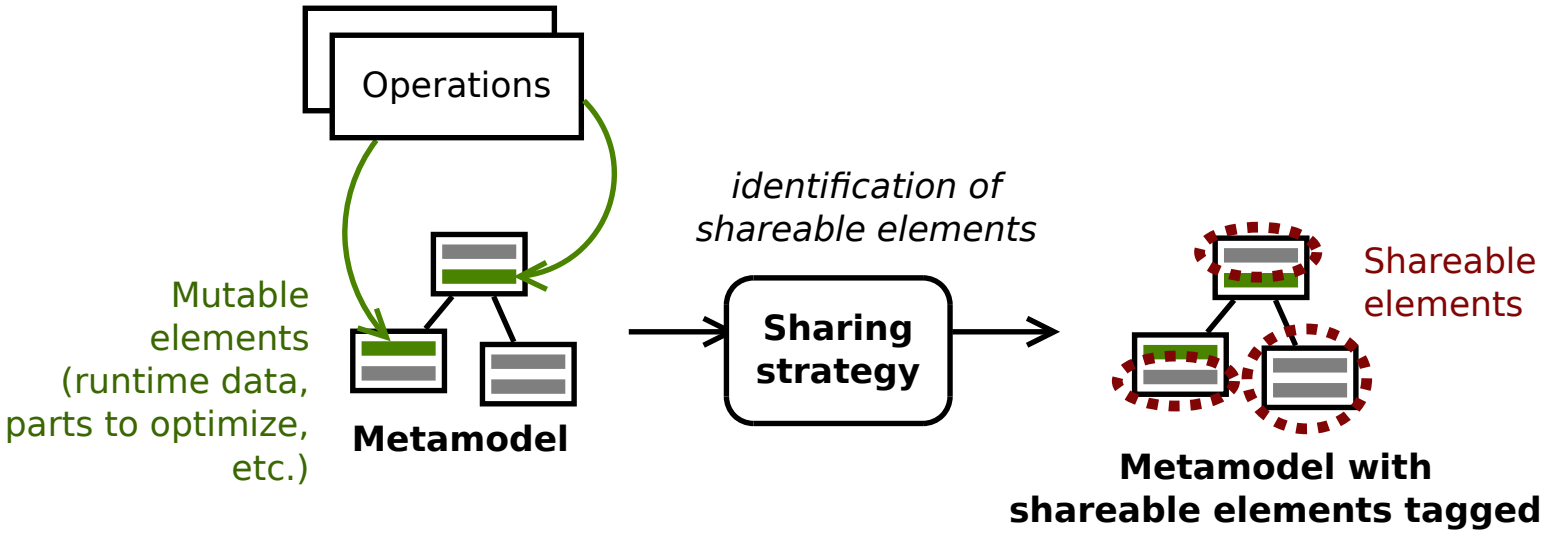
Approach: *static* identification of safely shareable parts

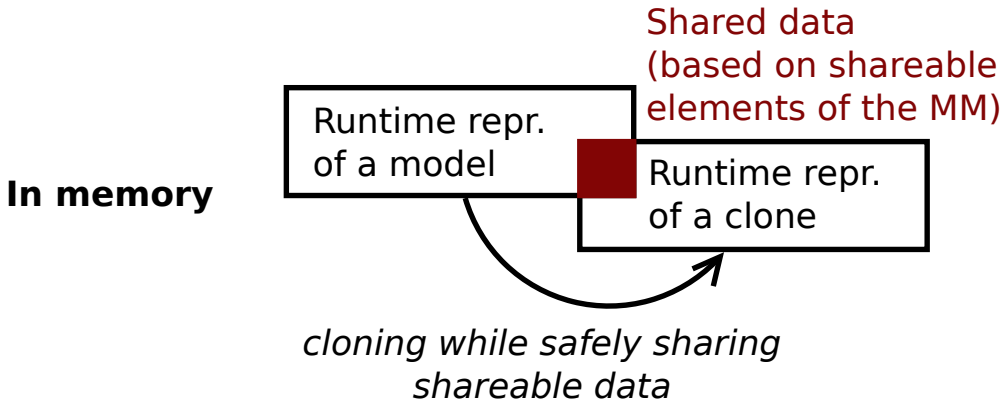


Approach: *static* identification of safely shareable parts

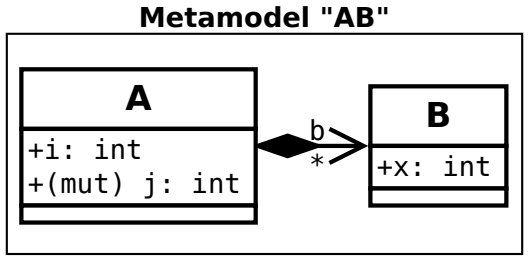


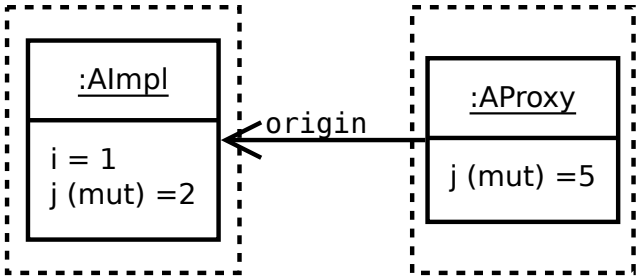
Approach: *static* identification of safely shareable parts



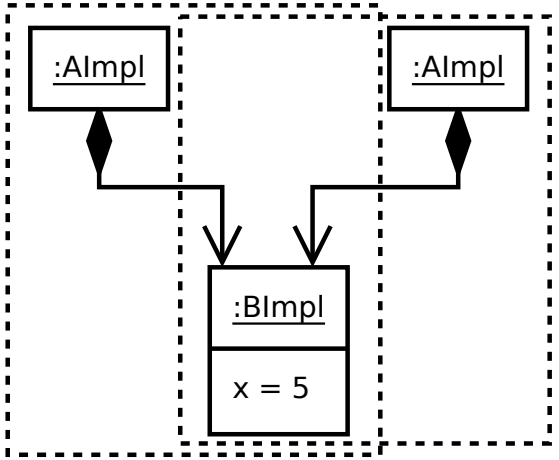


Shareable elements and sharing mechanisms





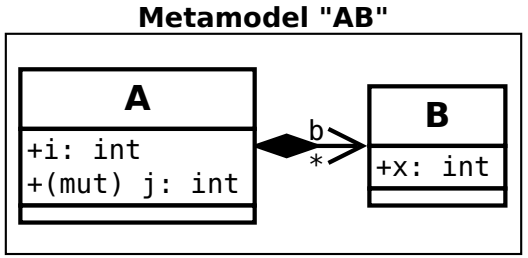
Field sharing using a partial proxy
(*i* considered shareable)

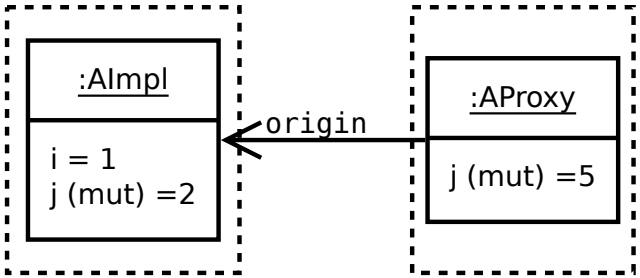


Object sharing
(*B* considered shareable)

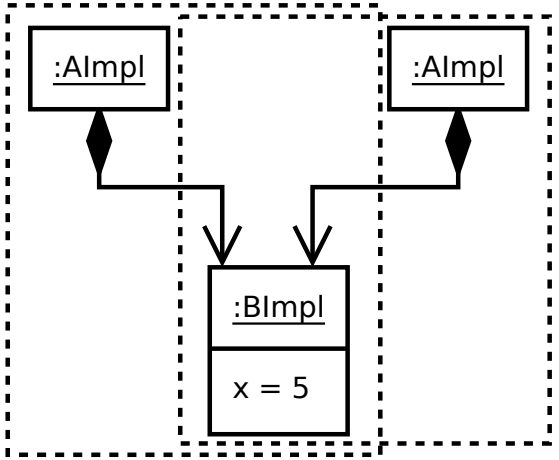
- Req #2 (efficiency) is not satisfied when sharing fields
- Req #4 (reflective layer) is not satisfied when sharing objects, since it breaks MOF `container()` operation

Shareable elements and sharing mechanisms





Field sharing using a partial proxy
(*i* considered shareable)



Object sharing
(*B* considered shareable)

- Req #2 (efficiency) is not satisfied when sharing fields
- Req #4 (reflective layer) is not satisfied when sharing objects, since it breaks MOF `container()` operation

Data sharing strategies

For design-time, **3 sharing strategies with trade-offs** between memory use and satisfaction of Req #2 and Req #4

DeepCloning	Nothing is shareable.
ShareFieldsOnly	Only immutable attributes are shareable.
ShareObjOnly	Classes that can't (transitively) access mutable parts are shareable.
ShareAll	Shareable elements are immutable attributes, classes whose properties are all shareable, and immutable references pointing to shareable classes.

For runtime, **1 generic algorithm** parameterized by a sharing strategy $\rightarrow$ *3 data sharing cloning operators*.

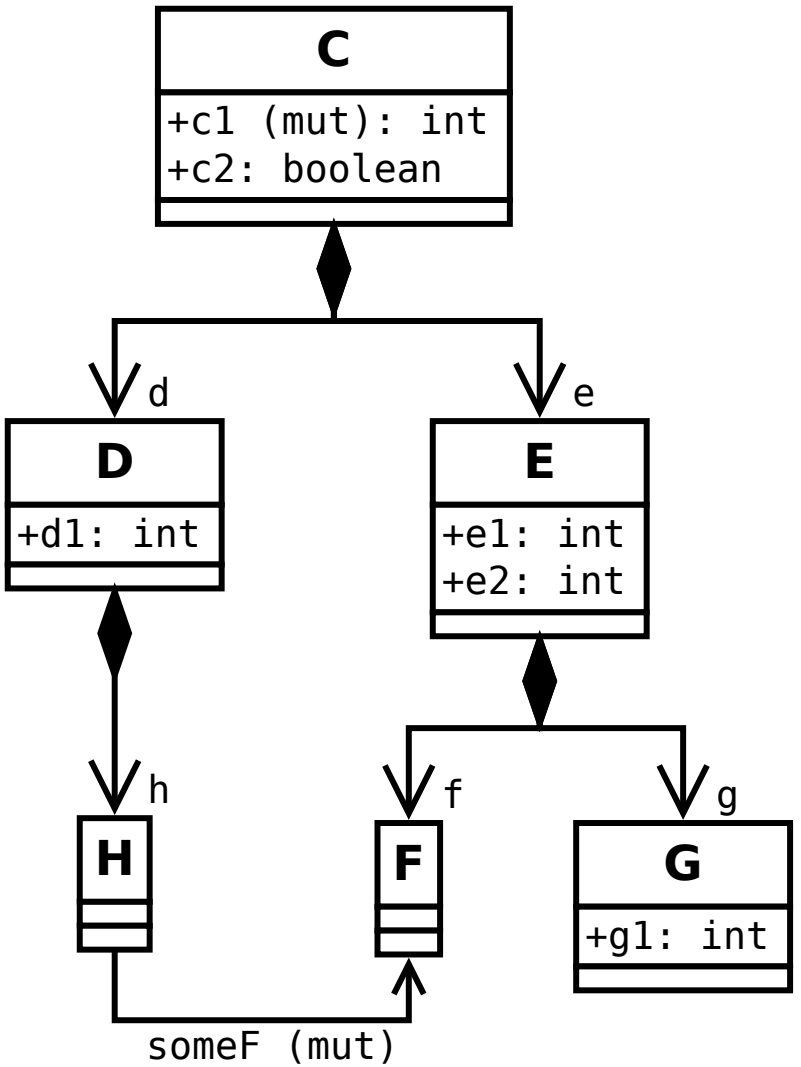
Data sharing strategies

For design-time, **3 sharing strategies with trade-offs** between memory use and satisfaction of Req #2 and Req #4

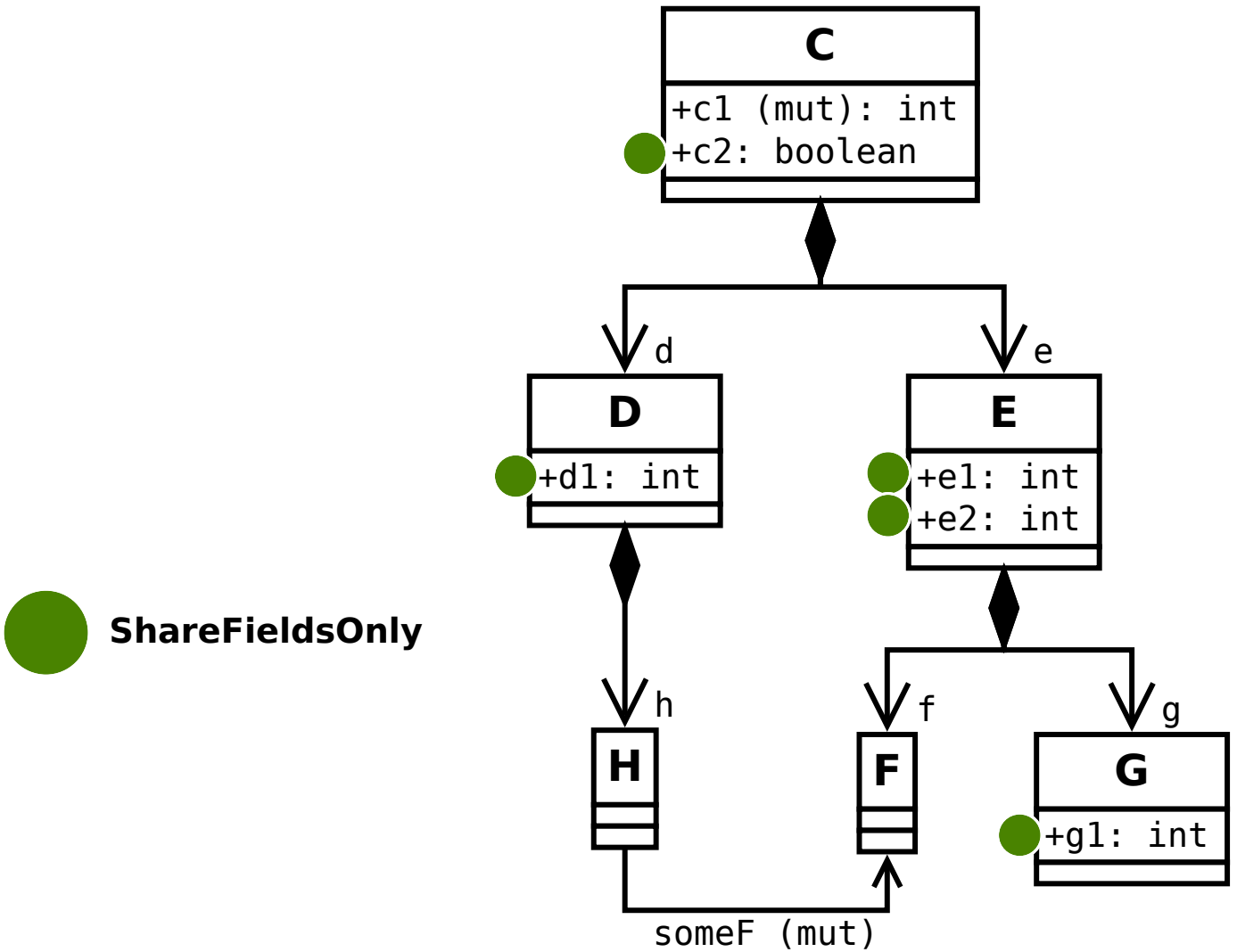
DeepCloning	Nothing is shareable.
ShareFieldsOnly	Only immutable attributes are shareable.
ShareObjOnly	Classes that can't (transitively) access mutable parts are shareable.
ShareAll	Shareable elements are immutable attributes, classes whose properties are all shareable, and immutable references pointing to shareable classes.

For runtime, **1 generic algorithm** parameterized by a sharing strategy $\rightarrow$ *3 data sharing cloning operators*.

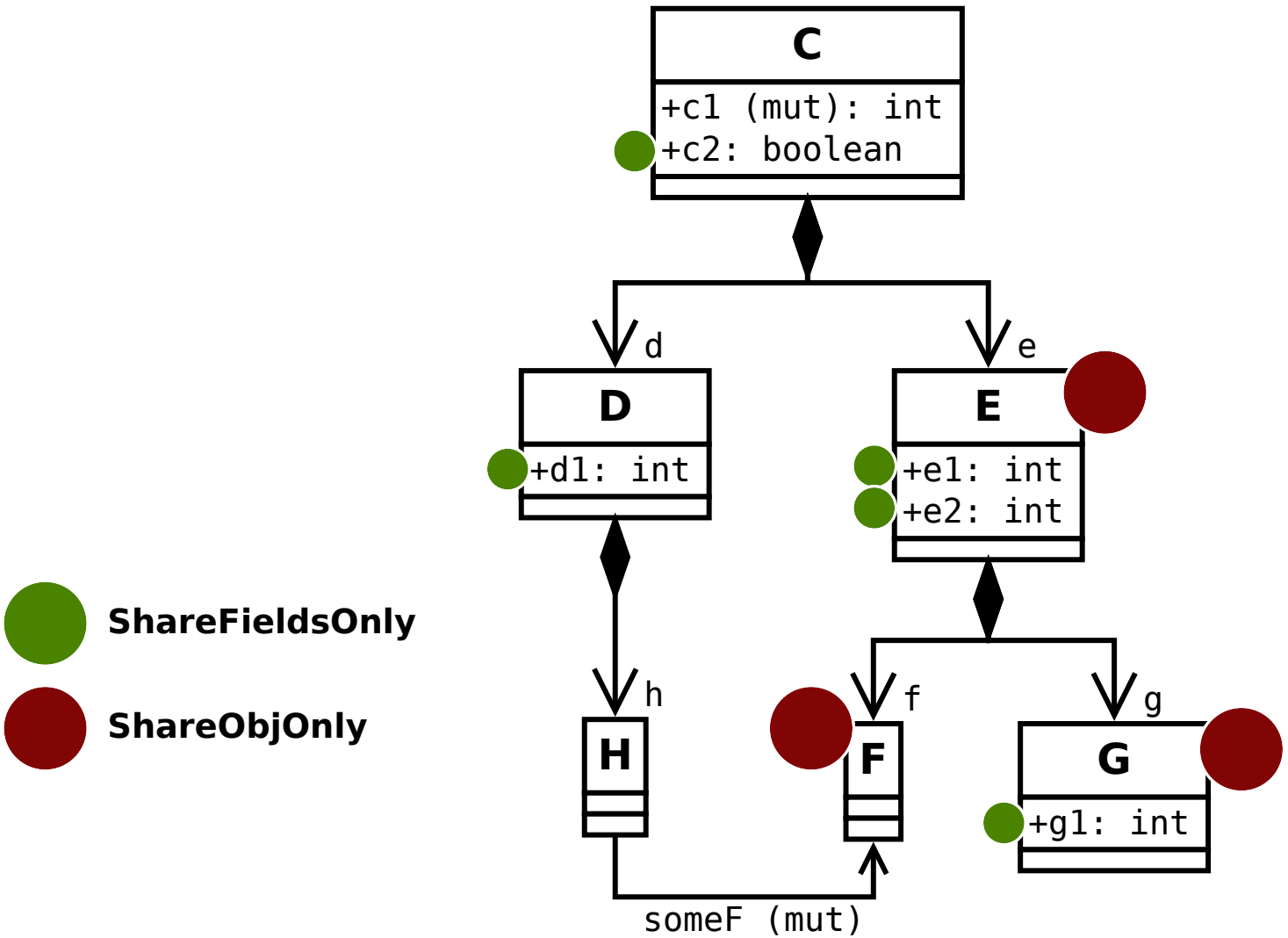
Data sharing strategies: example



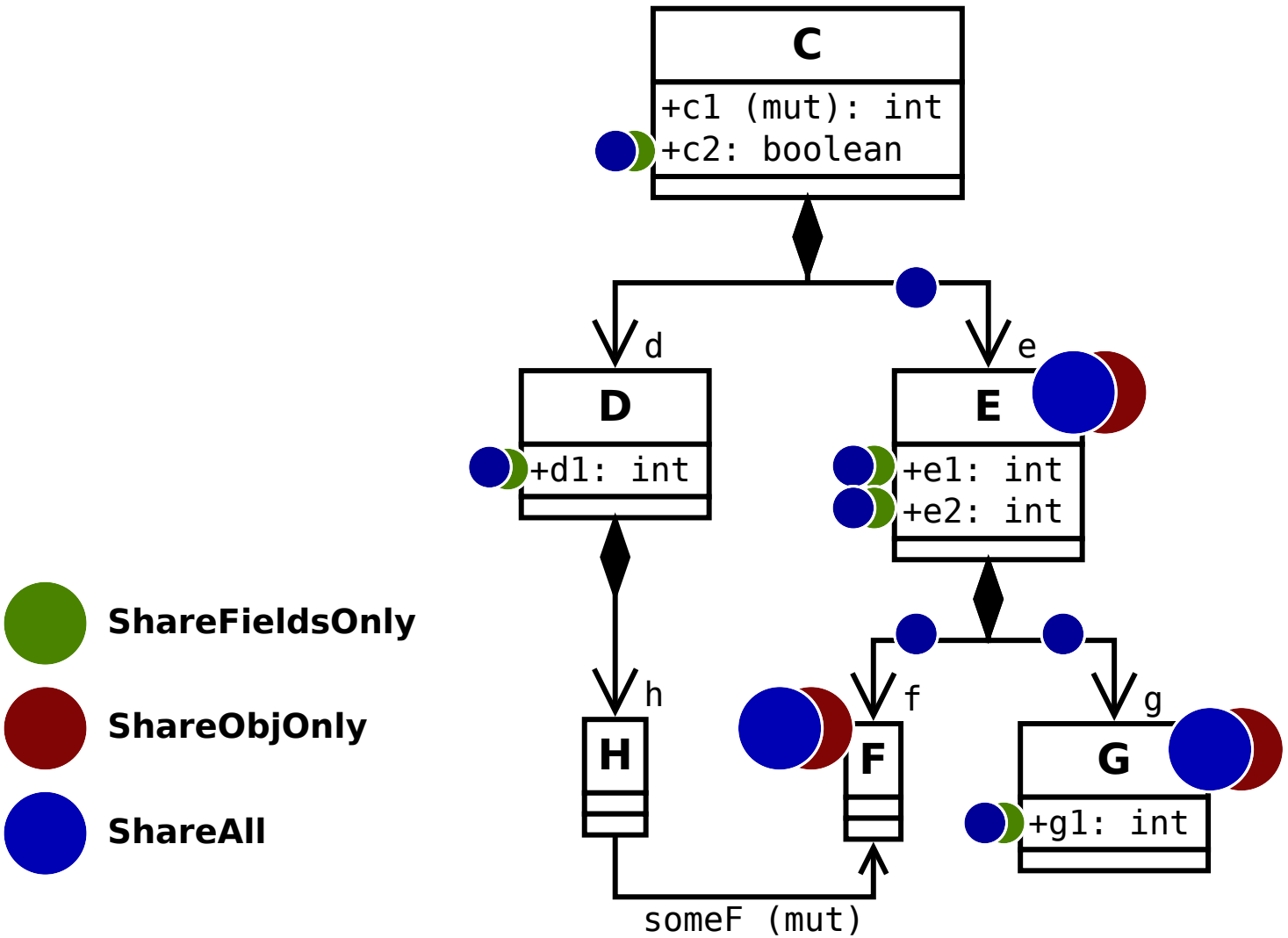
Data sharing strategies: example



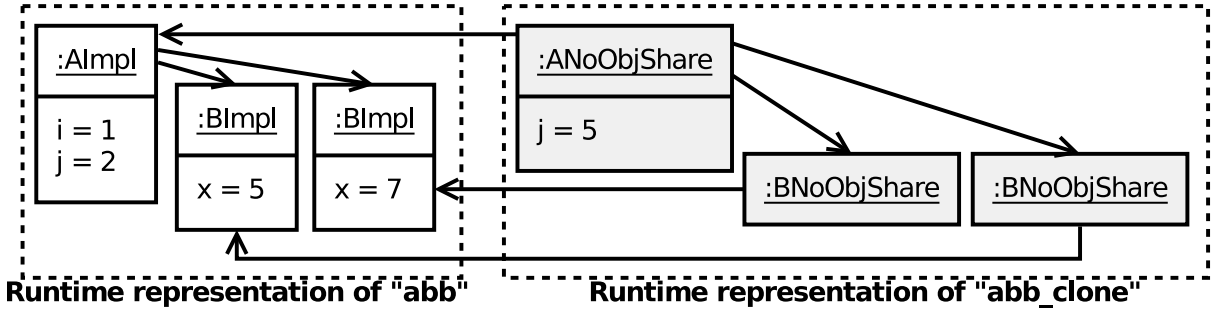
Data sharing strategies: example



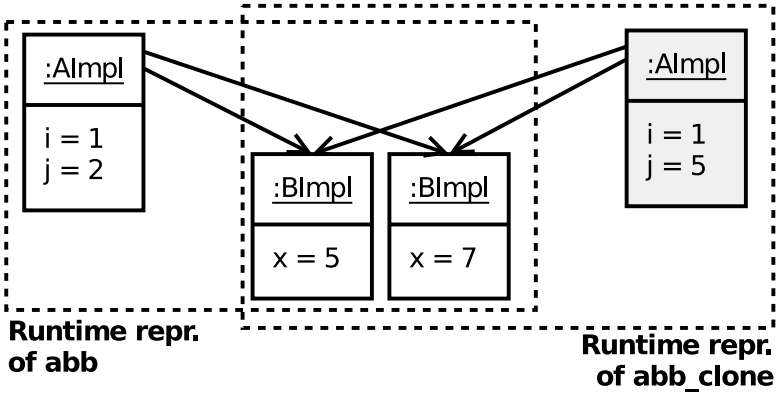
Data sharing strategies: example



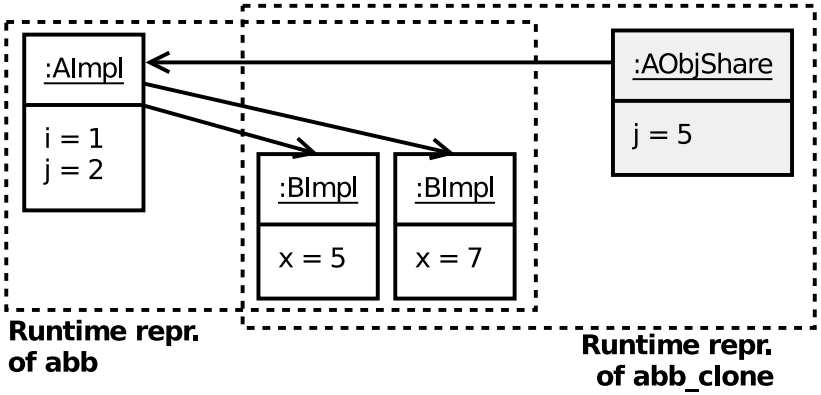
Resulting cloning operators



(a) *ShareFieldsOnly*



(b) *ShareObjOnly*



(c) *ShareAll*

Plan

Research questions

RQ#1

Do the new operators reduce the memory footprint of clones, compared to deep cloning?

RQ#2

Can a clone be manip. with the same efficiency as the original ?

RQ#3

Can a clone be manip. using the same generated API ?

RQ#4

Can a clone be manip. using the reflective layer (e.g. as stated in the MOF Reflection package)?

Evaluation – RQ#1 and RQ#2

Experiment

- **data set**: 100 randomly generated metamodels
- **memory measures**: gain as compared to *deep cloning*, after cloning the model 1000 times
- **performance measures**: loss of time as compared to the original model, when navigating 10 000 times through each object of the model while accessing all properties

Results

- **memory**: the more shareable parts, the more memory gain
- **performance**: worst median overhead is 9,5% when manipulating clones with fields sharing

Evaluation – RQ#1 and RQ#2

Experiment

- **data set**: 100 randomly generated metamodels
- **memory measures**: gain as compared to *deep cloning*, after cloning the model 1000 times
- **performance measures**: loss of time as compared to the original model, when navigating 10 000 times through each object of the model while accessing all properties

Results

- **memory**: the more shareable parts, the more memory gain
- **performance**: worst median overhead is 9,5% when manipulating clones with fields sharing

Evaluation – results overview

RQ1: memory
RQ2: efficiency
RQ3: same API
RQ4: reflective layer

	RQ1	RQ2	RQ3	RQ4
DeepCloning	✗	✓	✓	✓
ShareFieldsOnly	+	- -	✓	✓
ShareObjOnly	++	✓	✓	✗
ShareAll	+++	-	✓	✗

Plan

Conclusion

- Model cloning is required in many kinds of applications, but *deep cloning* not scalable
- Approach: **find shareable parts at the metamodel level**, then share both runtime objects and fields between runtime representations of clones
- 3 data sharing strategies + 1 algorithm = 3 cloning operators
- Evaluation shows memory gain with tradeoffs regarding efficiency and/or reflective layer compatibility

Possible future work

Automate the choice or operator using static analysis (e.g. if `eContainer()` is used, then some operators are disabled)

Done!

Thank you for your attention ☺

Tool (Eclipse plugin):

`http://moclodash.gforge.inria.fr/`