Architecting
Domain-Specific Languages

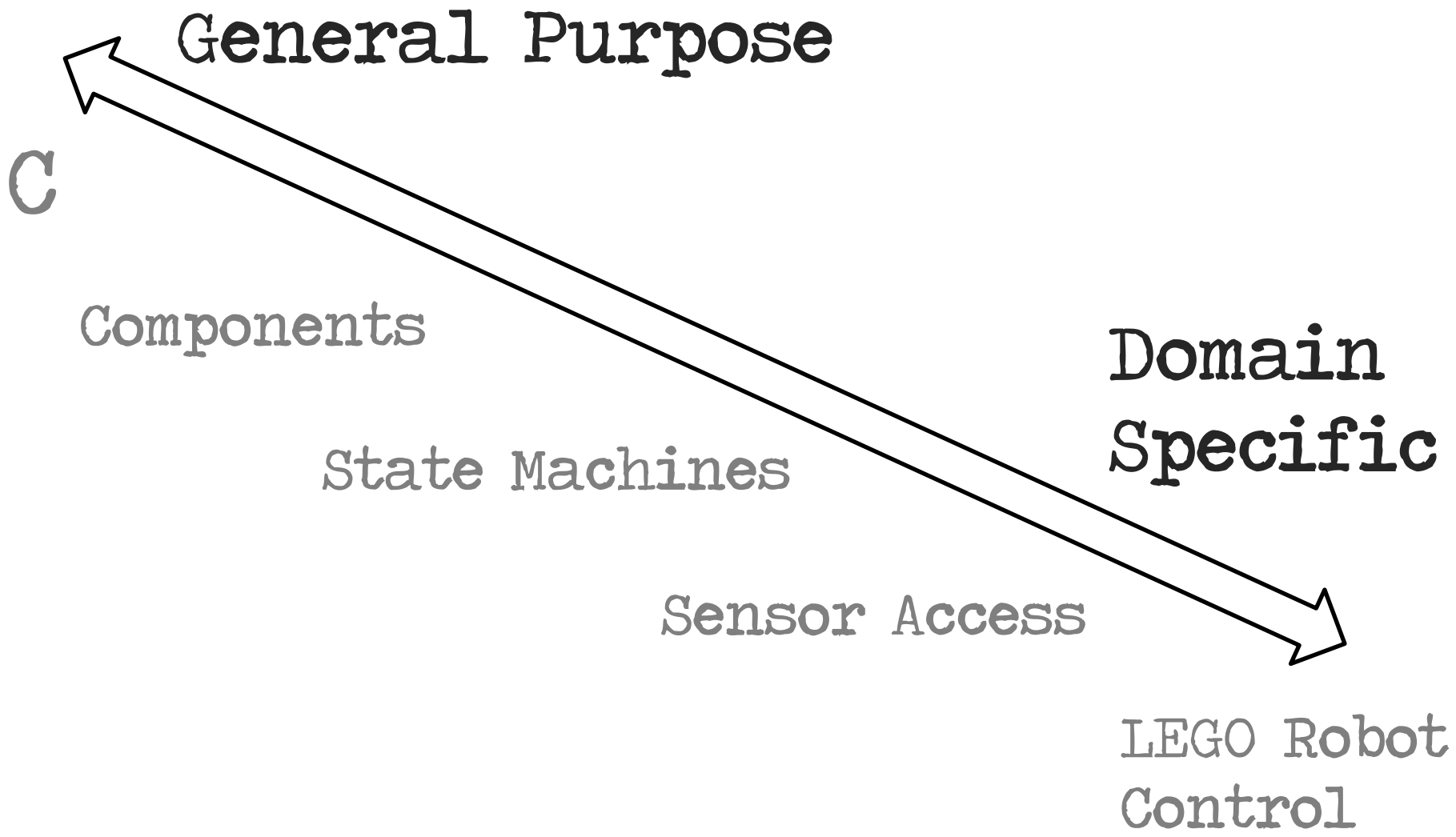
Markus Völter

voelter@acm.org
www.voelter.de
@markusvoelter

0

Introduction

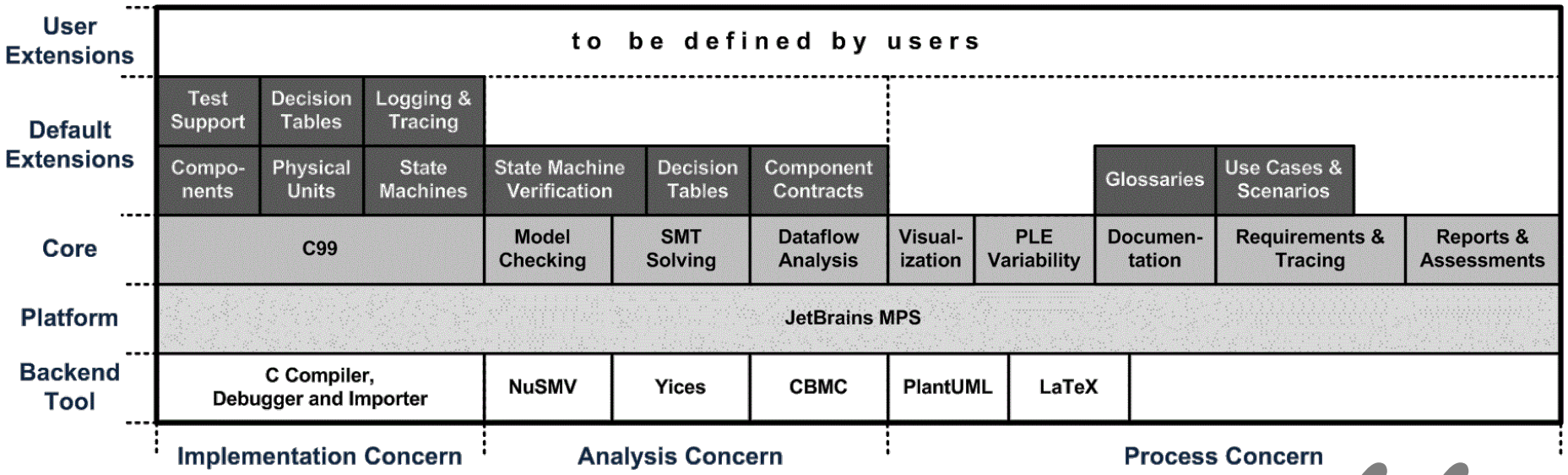
	more in GPLs	more in DSL
Domain Size	large and complex	smaller and well-defined
Designed by	guru or committee	a few engineers and domain experts
Language Size	large	small
Turing-completeness	almost always	often not
User Community	large, anonymous and widespread	small, accessible and local
In-language abstraction	sophisticated	limited
Lifespan	years to decades	months to years (driven by context)
Evolution	slow, often standardized	fast-paced
Incompatible Changes	almost impossible	feasible


General Purpose
C
Components
Domain Specific
State Machines
Sensor Access
LEGO Robot Control

1

Case Study: mbeddr

An extensible set of integrated languages for embedded software engineering.



“Specific Languages”

mbeddr

StateMachines – tutorial – [~/Documents/mbeddr/mbeddr.core/code/applications/tutorial]

```
#constant TAKEOFF = 100; -> implements PointsForTakeoff
#constant HIGH_SPEED = 10; -> implements FasterThan100
#constant VERY_HIGH_SPEED = 20; -> implements FasterThan200
#constant LANDING = 100; -> implements FullStop

[verifiable]
exported statemachine FlightAnalyzer initial = beforeFlight {
  in event next(Trackpoint* tp) <no binding>
  in event reset() <no binding>
  out event crashNotification() => raiseAlarm
  readable var int16 points = 0
  state beforeFlight {
    //[ Here is a comment on a transition. ]
    on next [tp->alt == 0 m] -> airborne
    [exit { points += TAKEOFF; }] -> implements PointsForTakeoff
  } state bef
  state airborne {
    on next [tp->alt == 0 m && tp->speed == 0] -> crashed
    on next [tp->alt == 0 m && tp->
    [on next [tp->speed > 200 mps &
    [on next [tp->speed > 100 mps &
    on reset [ ] -> beforeFlight
  } state airborne
  state landing {
    on next [tp->speed == 0 mps] -> landed
    [on next [tp->speed > 0 mps] -> landing { points--; }] -> imp
```

Error: type int16/[m / s] is not comparable with (uint8 || int8)

```
alt                    ^DataStructures.Trackpoint.alt (Member)
crashNotification      ^StateMachines.FlightAnalyzer.crashNotification (OutEvent)
id                     ^DataStructures.Trackpoint.id (Member)
speed                  ^DataStructures.Trackpoint.speed (Member)
time                   ^DataStructures.Trackpoint.time (Member)
x                      ^DataStructures.Trackpoint.x (Member)
y                      ^DataStructures.Trackpoint.y (Member)
```

```
#constant TAKEOFF = 100; -> implements PointsForTakeoff
#constant HIGH_SPEED = 10; -> implements FasterThan100
#constant VERY_HIGH_SPEED = 20; -> implements FasterThan200
#constant LANDING = 100; -> implements FullStop

[verifiable]
exported statemachine FlightAnalyzer initial = beforeFlight
```

	next(Trackpoint* tp)
beforeFlight	//[Here is a comment on a transition.]
	[tp->alt == 0 m] -> airborne
airborne	[tp->alt == 0 m && tp->speed == 0] -> crashe
	[tp->alt == 0 m && tp->speed > 0 mps] -> lan
	[tp->speed > 200 mps && tp->alt == 0 m] ->
	[tp->speed > 100 mps && tp->speed <= 200 mp
	tp->alt == 0 m] -> airborne
landing	[tp->speed == 0 mps] -> landed
	[tp->speed > 0 mps] -> landing -> implements Sh
landed	

```
alyzer initial = b
```

	next(Trackpoi
beforeFlight	[tp->alt > 0
composite state airborne initial = flying {	[onTheGround

Open Source @ eclipse.org
Eclipse Public License 1.0
http://mbeddr.com

itemis France: **Smart Meter**

First significant mbeddr project
ca. 100,000 LoC
about to be finished
great modularity due to components
uses physical units extensively
great test coverage due to special extensions

ACCEnT Control.Lab

LMS INTERNATIONAL
Researchpark Haasrode 1237 | Interleuvenlaan 68 | B-3001 Leuven [Belgium]
T +32 16 384 200 | F +32 16 384 350 | info@lmsintl.com | www.lmsintl.com

Worldwide
For the address of your local representative, please visit www.lmsintl.com/lmsworldwide

LMS is a leading provider of test and mechatronic simulation software and engineering services in the automotive, aerospace and other advanced manufacturing industries. As a business segment within Siemens PLM Software, LMS provides a unique portfolio of products and services for manufacturing companies to manage the complexities of tomorrow's product development by incorporating model-based mechatronic simulation and advanced testing in the product development process. LMS tunes into mission-critical engineering attributes, ranging from system dynamics, structural integrity and sound quality to durability, safety and power consumption. With multi-domain and mechatronic simulation solutions, LMS addresses the complex engineering challenges associated with intelligent system design and model-based systems engineering. Thanks to its technology and more than 1250 dedicated people, LMS has become the partner of choice of more than 5000 manufacturing companies worldwide. LMS operates in more than 30 key locations around the world.

Siemens PLM Software

SIEMENS

LMS®

A Siemens Business

mbeddr

20+ Projects in various stages

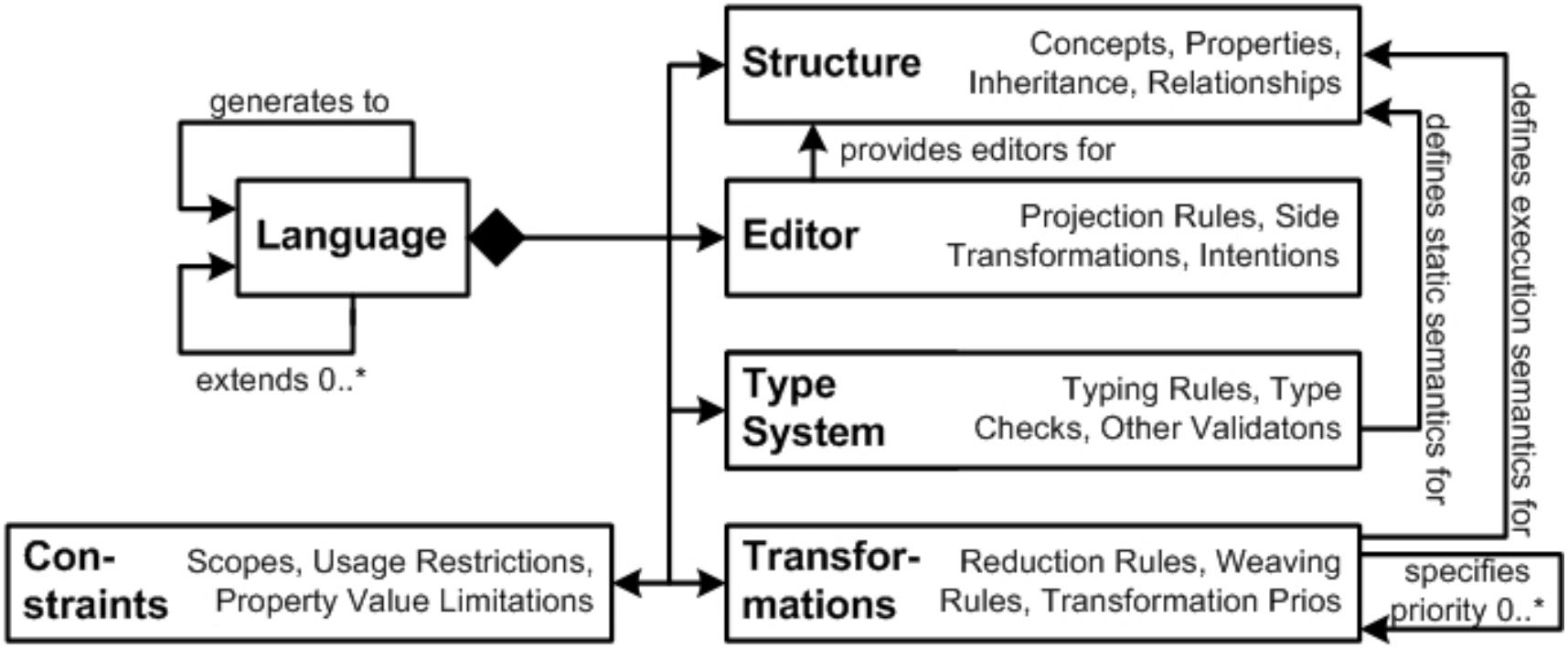
by various “Big Name” companies.

Branching into other domains

insurance, financial, tax

Open Source
Apache 2.0
http://jetbrains.com/mps

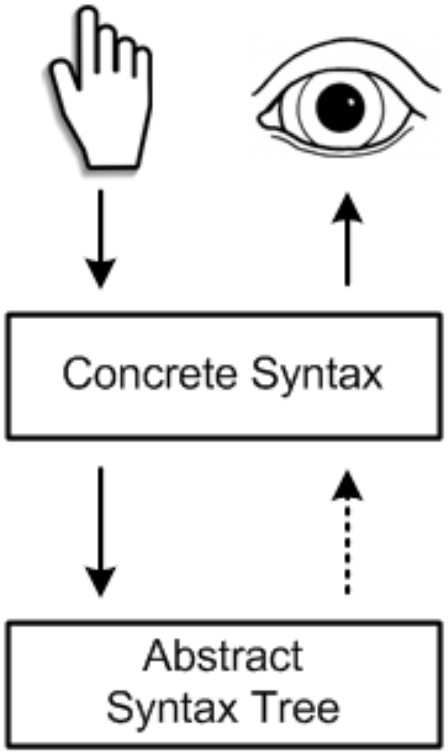
[Language Workbench]



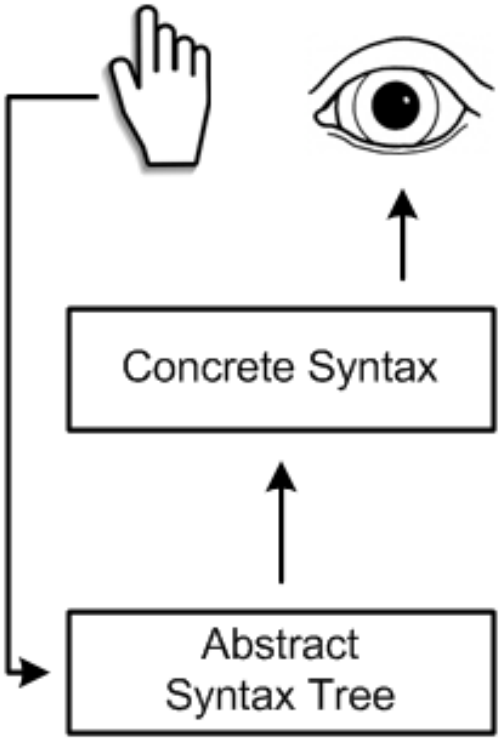
+ Refactorings, Find Usages, Syntax Coloring, Debugging, ...

[Projectional Editing]

Parsing



Projectional Editing



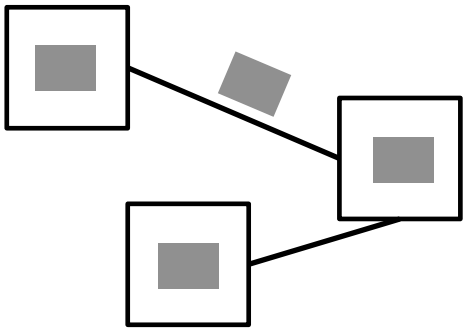
[Projectional Editing]
Syntactic Flexibility

Regular Code/Text

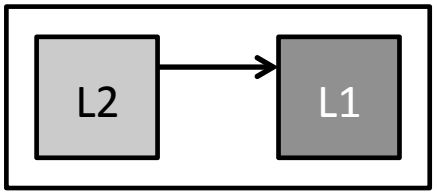
Mathematical

Tables

Graphical



[Projectional Editing]
Language Composition



Separate Files

Type System
Transformation
Constraints

In One File

Type System
Transformation
Constraints
Syntax
IDE

2

Expressivity

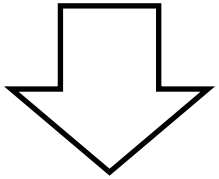
Shorter Programs

More Accessible Semantics

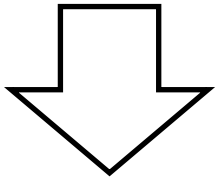
For a limited Domain!

Domain Knowledge encapsulated in language

Smaller Domain



More Specialized Language



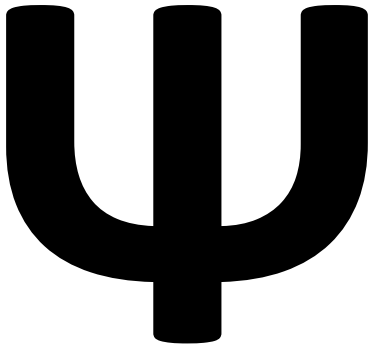
Shorter Programs

The do-what-I-want language

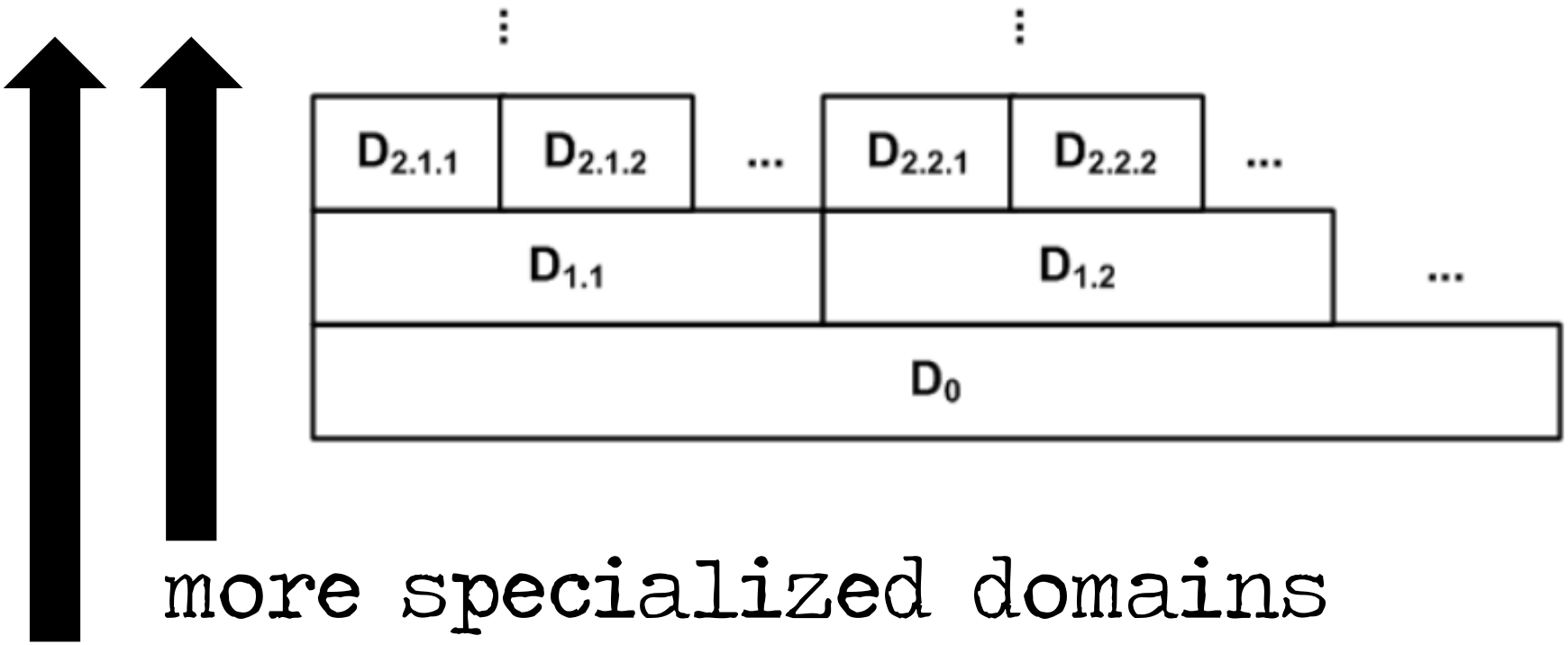
Ψ

Single Program vs. Class/Domain

No Variability!

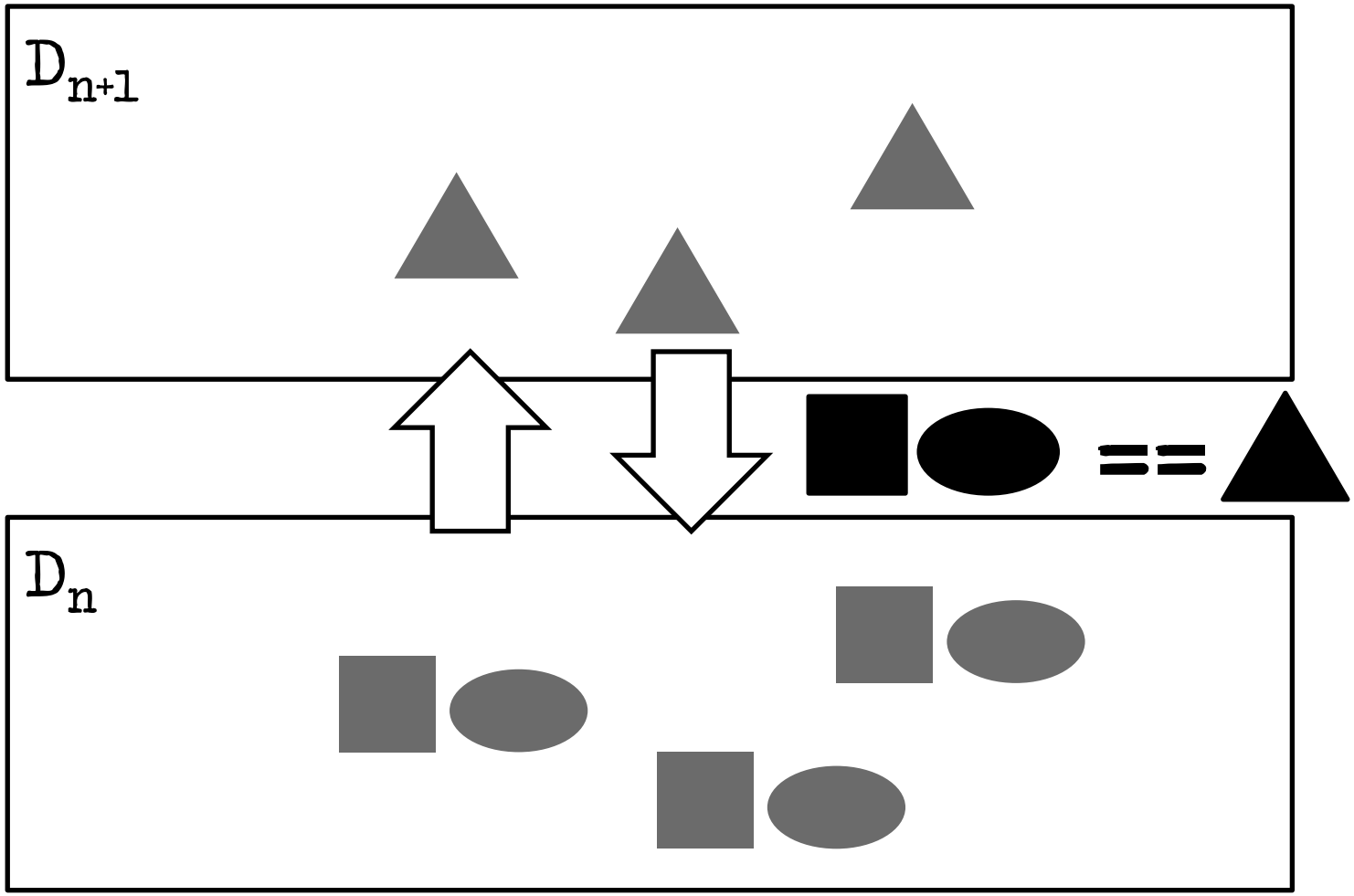


Domain Hierarchy



more specialized domains
more specialized languages

Reification



Reification

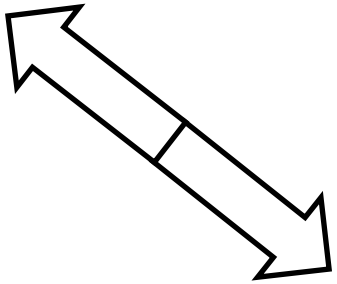
Transformation/
Generation

Language
Definition

C Extensions

Requirements

Linguistic Abstraction



In-Language Abstraction

Libraries
Classes
Frameworks

Linguistic Abstraction

Analyzable
Better IDE Support

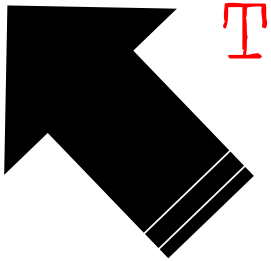
In-Language Abstraction

User-Definable
Simpler Language

Functions
Components

Linguistic Abstraction

Analyzable
Better IDE Support



Special Treatment!

In-Language Abstraction

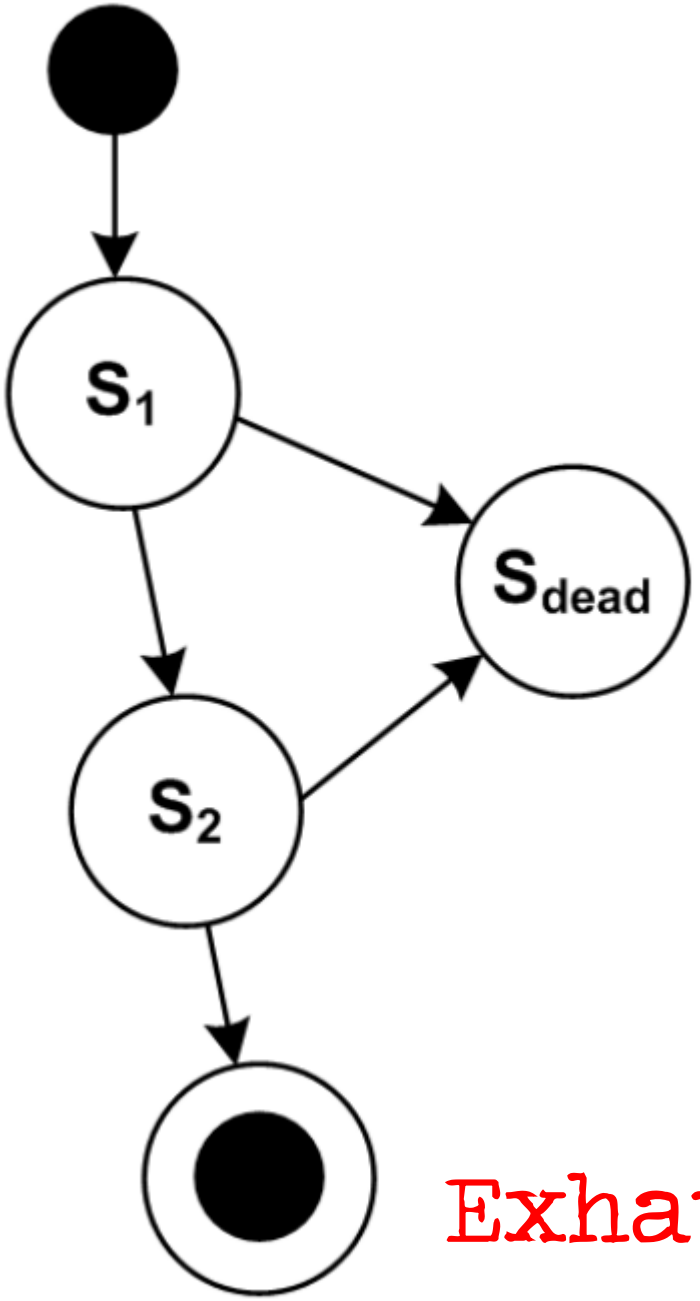
User-Definable
Simpler Language

Linguistic Abstraction

Std Lib

In-Language Abstraction

Block Library



Unique State Names

Unreachable States

Dead End States

Guard Decidability

Reachability

Exhaustive Search, Proof!

Component Verification
SM Model Checking

3

Notation

UI for the language!

Important for acceptance by users!

- Textual
- Prose
- Symbolic/Math
- Tabular
- Graphical

Reuse existing syntax of domain, if any!

Tools let you freely combine all kinds.

Reuse existing syntax of domain, if any!

mbeddr

State Machine Tables
Math
Component Wiring
Prose
Traces

4

Type System

Static Semantics

Execution Semantics

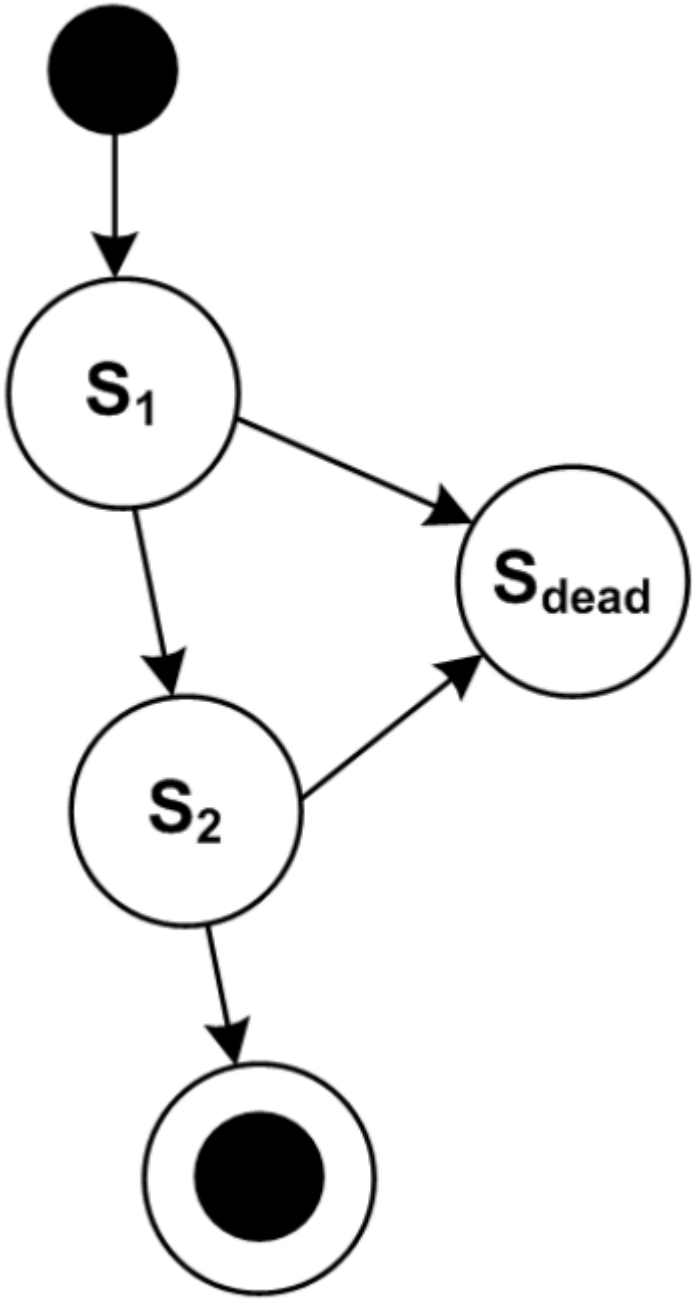
Static Semantics

Execution Semantics

Static Semantics

Constraints

Type Systems



Unique State Names

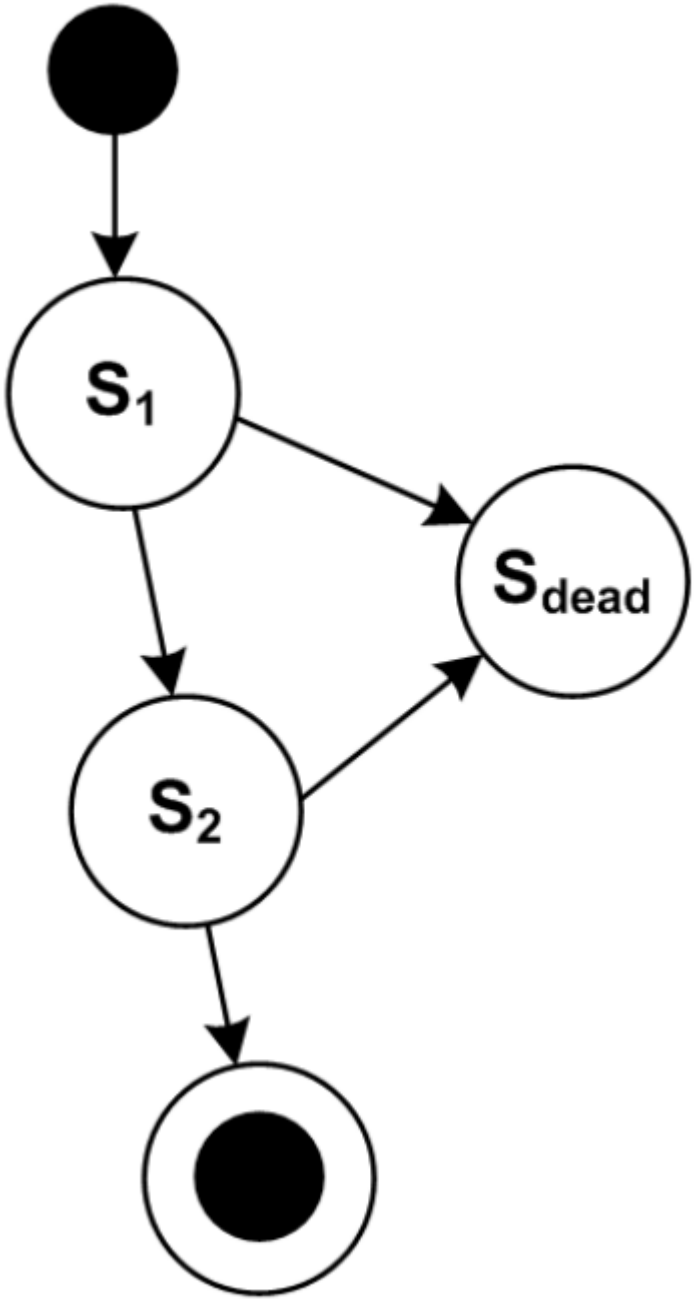
Unreachable States

Dead End States

...

Example

Exten
ded C



Unique State Names

Unreachable States

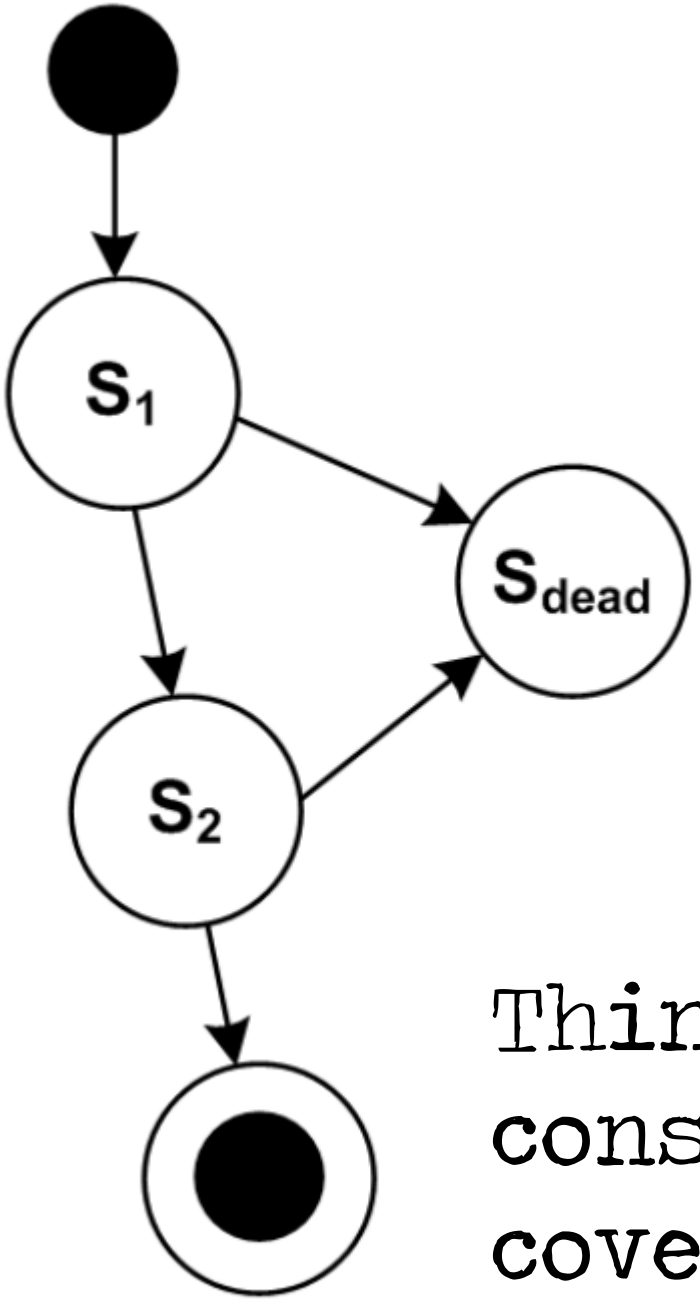
Dead End States

...

Easier to do on a declarative Level!

Example

Extended C



Unique State Names

Unreachable States

Dead End States

...

Easier to do on a declarative Level!

Thinking of all constraints is a coverage problem!

Example

Exten
ded C

```
var int x = 2 * someFunction(sqrt(2));
```

Assign fixed types

Derive Types

Calculate Common Types

Check Type Consistency

What does a type system do?

Intent + Check

```
var int x = 2 *
    someFunction(sqrt(2));
```

Derive

```
var x = 2 * some
    Function(sqrt(2));
```

More code

Better error messages

Better Performance

More convenient

More complex checkers

Harder to understand for users

Classical C Types
Closures

5

Execution

Def: Semantics

… via mapping to lower level

$$semantics(p_{L_D}) := q_{L_{D-1}}$$

$$where\ OB(p_{L_D}) == OB(q_{L_{D-1}})$$

OB: Observable Behaviour (Test Cases)

Def: Semantics

… via mapping to lower level

$$semantics(p_{L_D}) := q_{L_{D-1}}$$

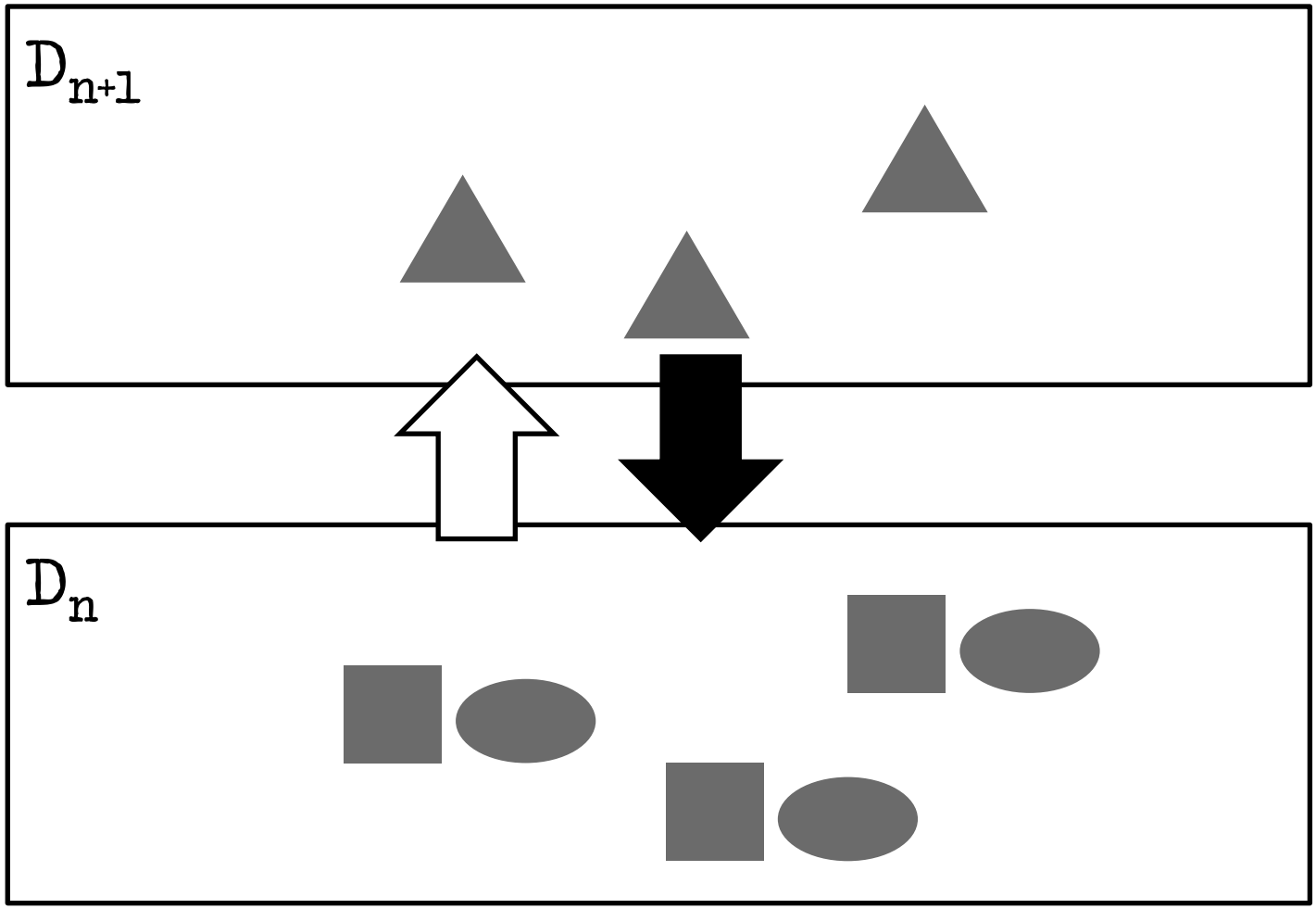
$$where\ OB(p_{L_D}) == OB(q_{L_{D-1}})$$

L_D

$\Downarrow$ Transformation
Interpretation

L_{D-1}

Transformation



Transformation

L_D

$\Downarrow$ Transformation

L_{D-1} Known Semantics!

Transformation

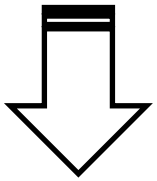
L_D

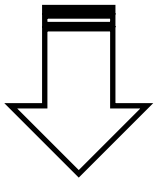
Correct!? $\Downarrow$ Transformation

L_{D-1}

Transformation

Tests (D) L_D





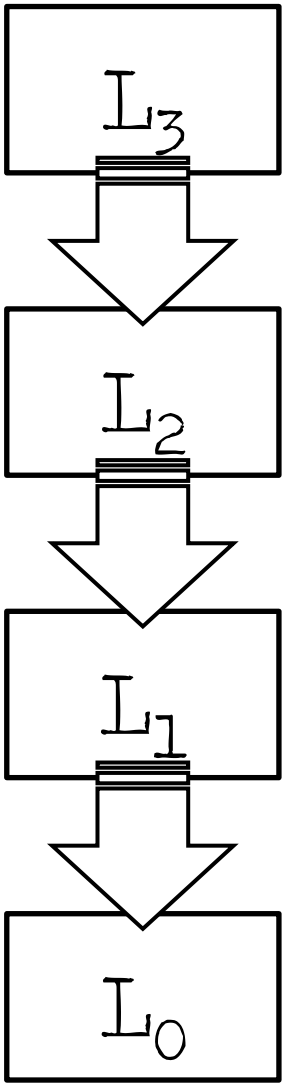
Transformation

Tests (D-1) L_{D-1}

Run tests on both levels; all pass.
Coverage Problem!

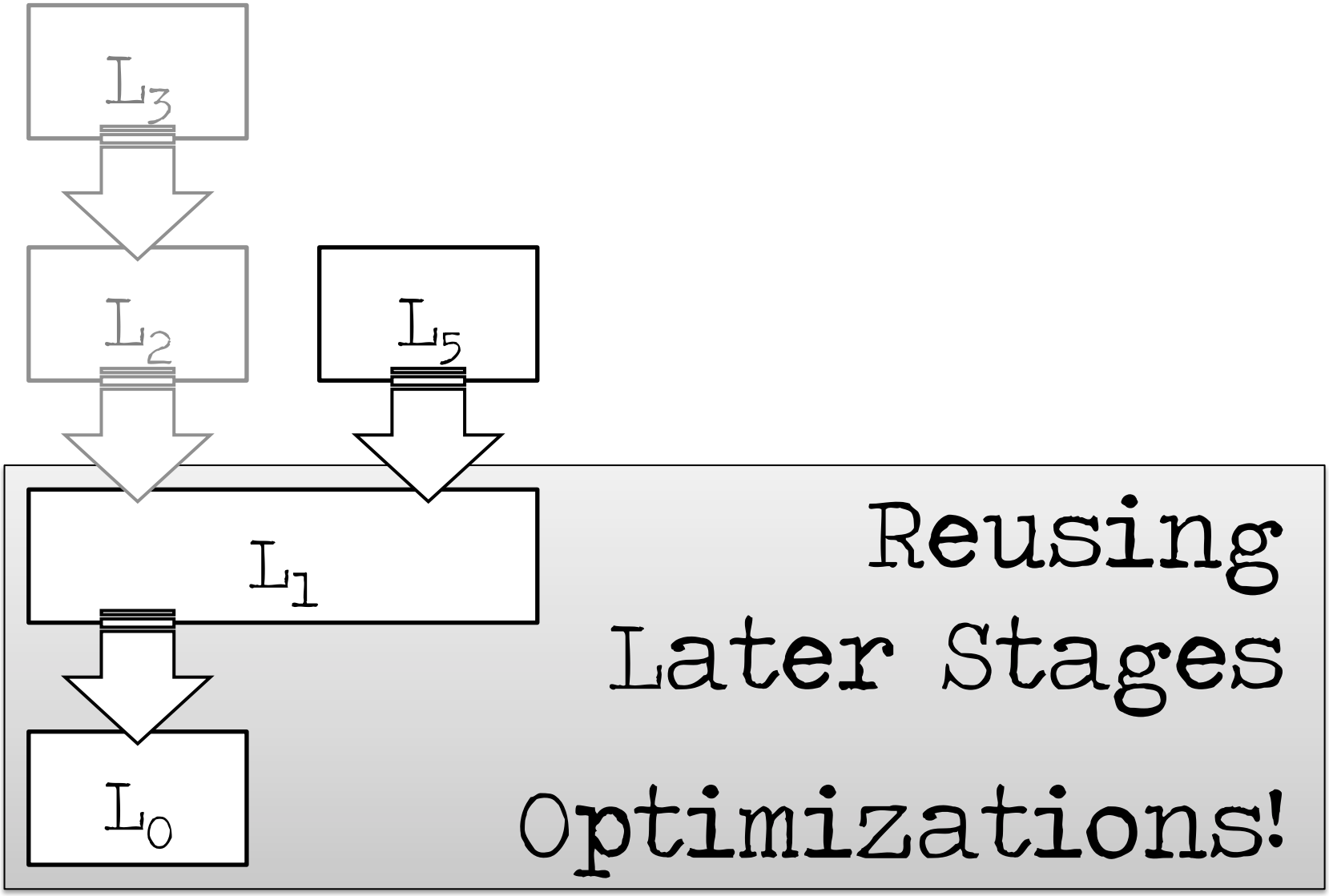
mbeddr Tests

Multi-Stage

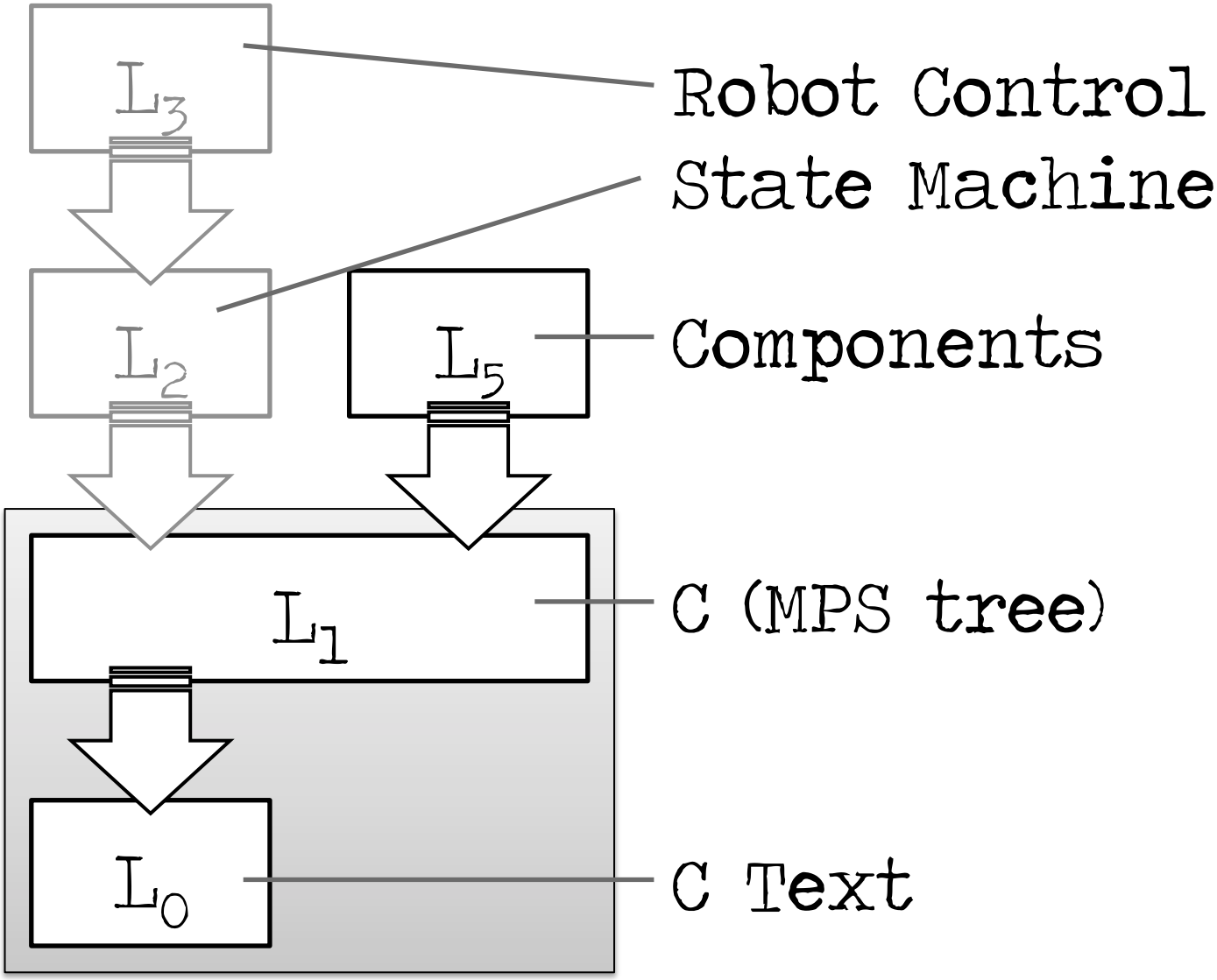


Modularization

Multi-Stage: Reuse



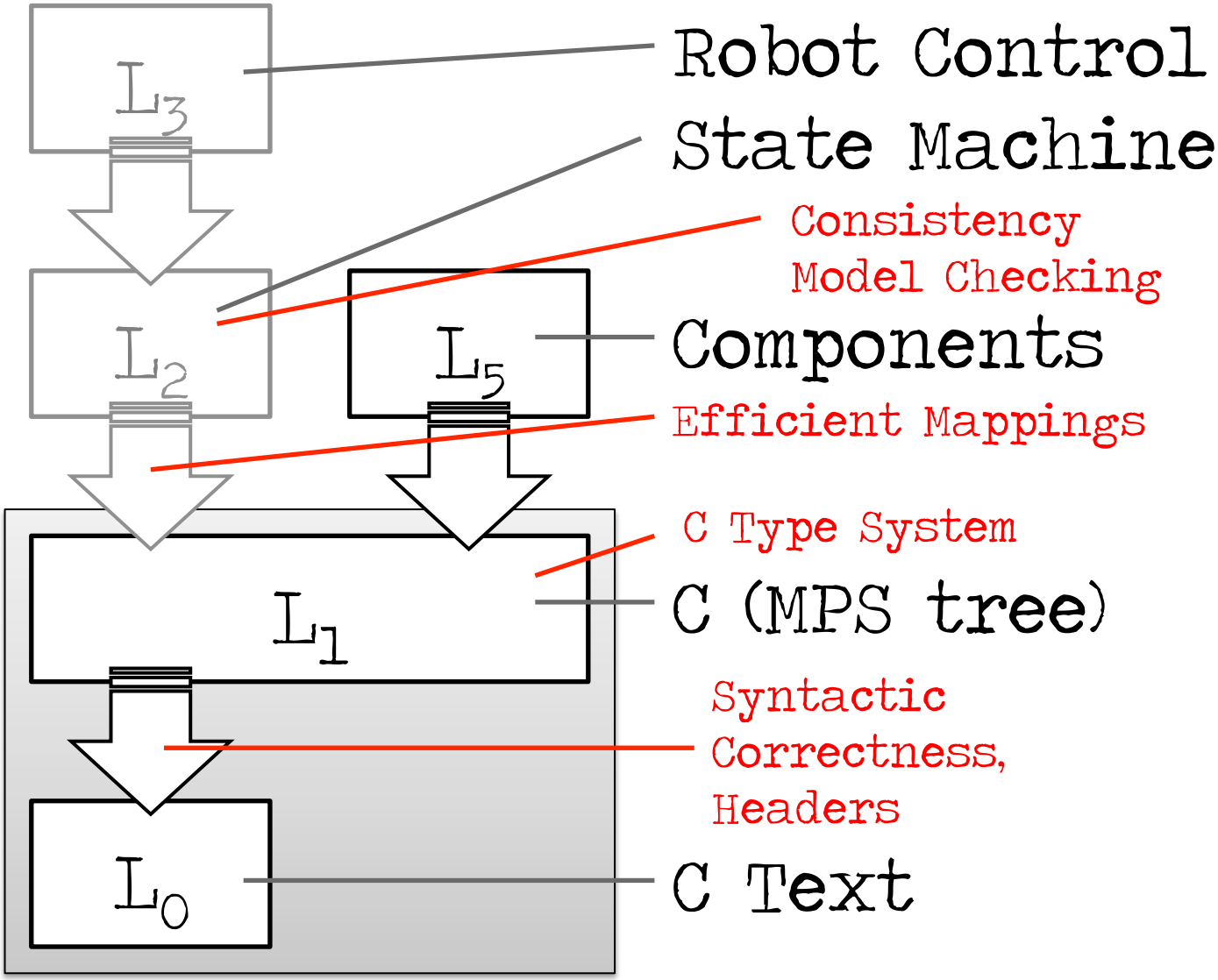
Multi-Stage: Reuse



Example

Exten
ded C

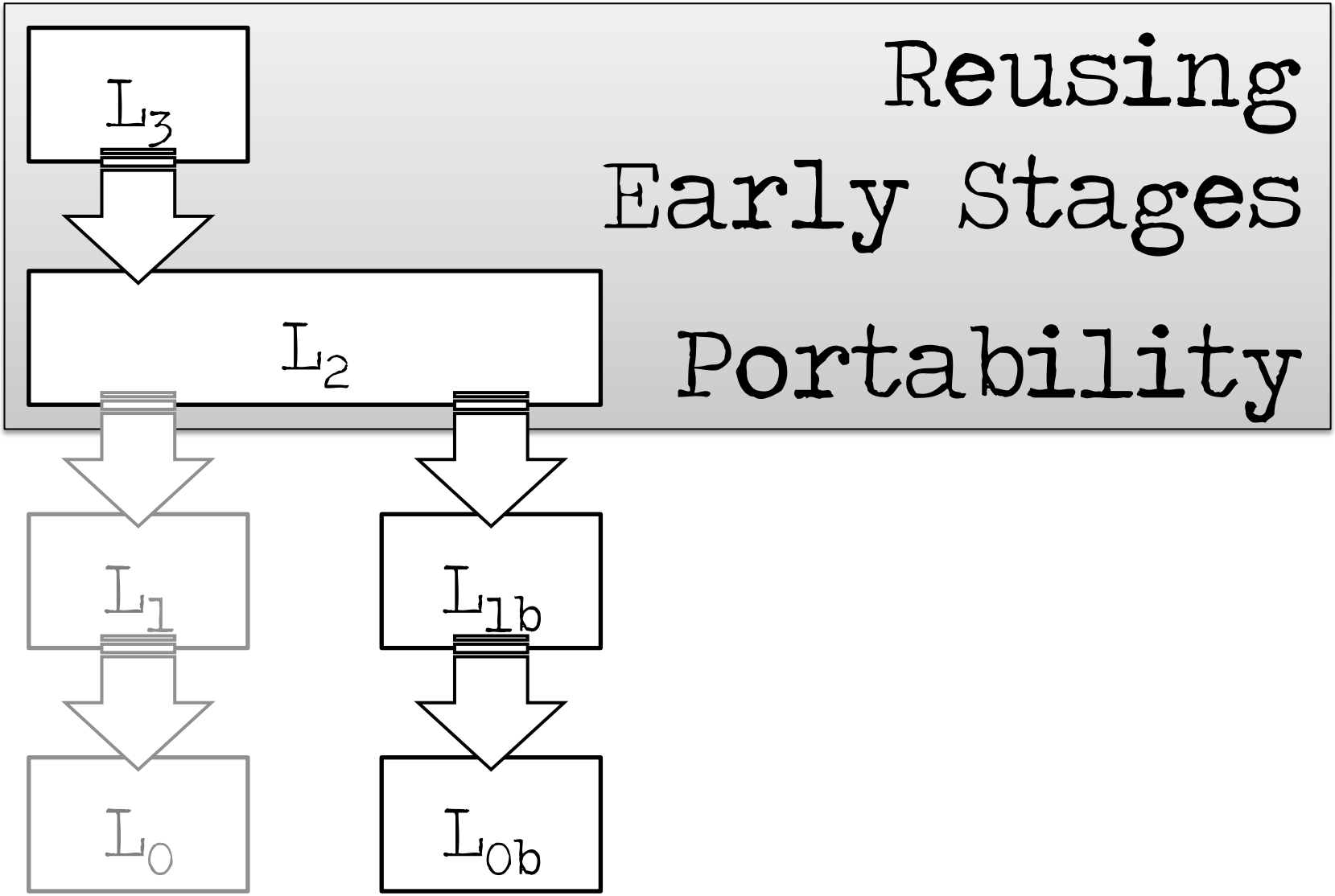
Multi-Stage: Reuse



Example

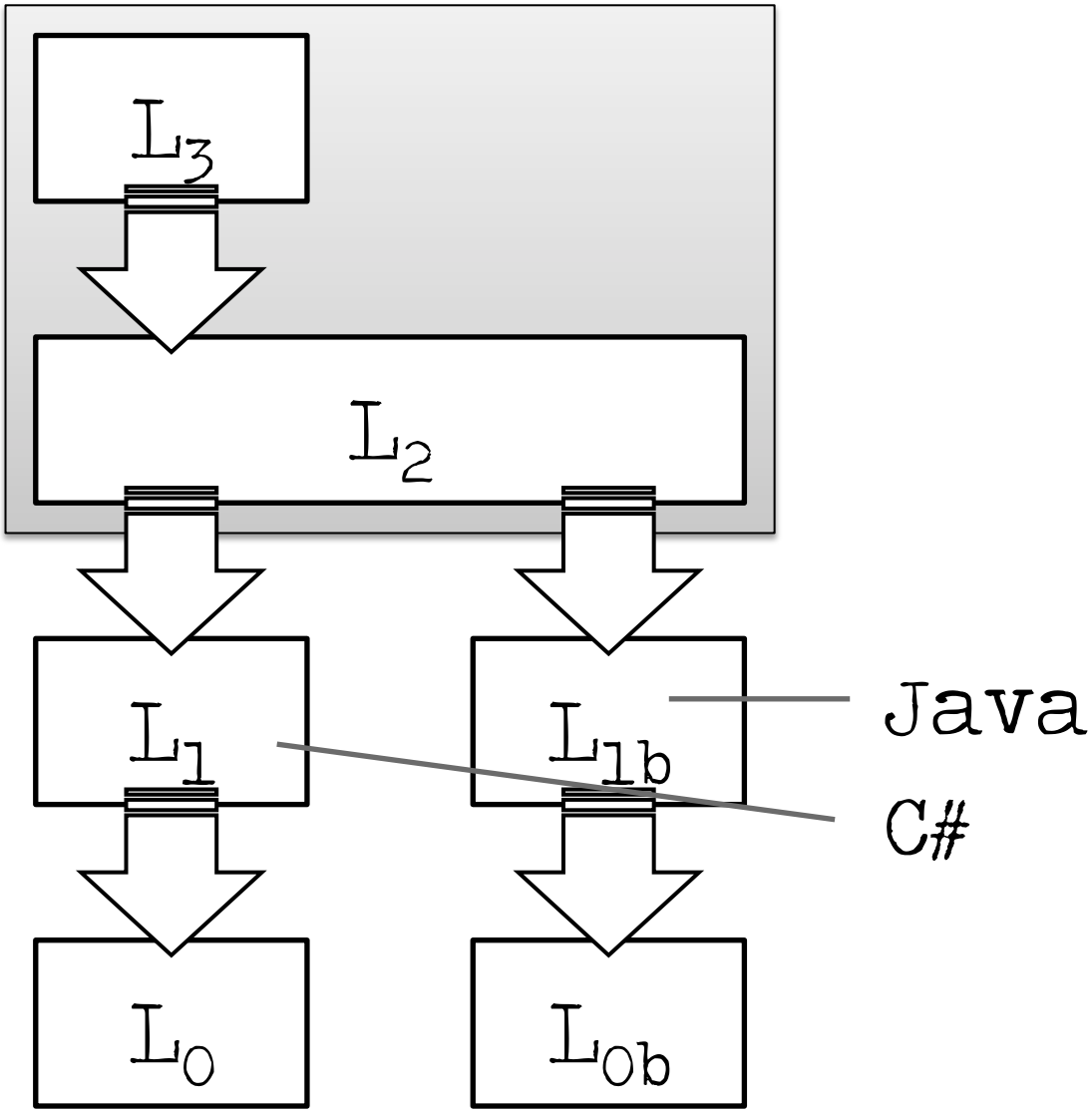
Exten
ded C

Multi-Stage: Reuse



Reusing
Early Stages
Portability

Multi-Stage: Reuse

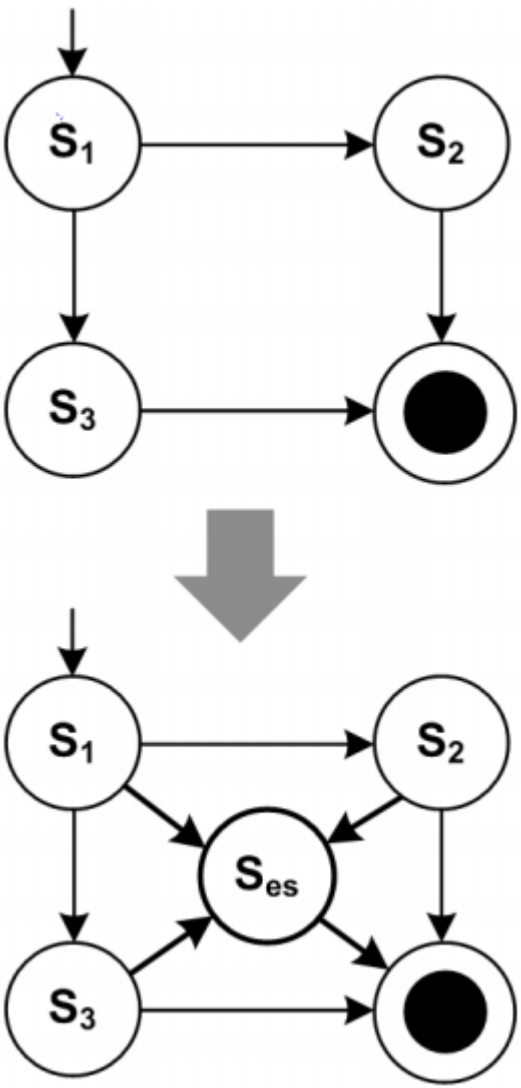


Example

Extended C

Mock Components

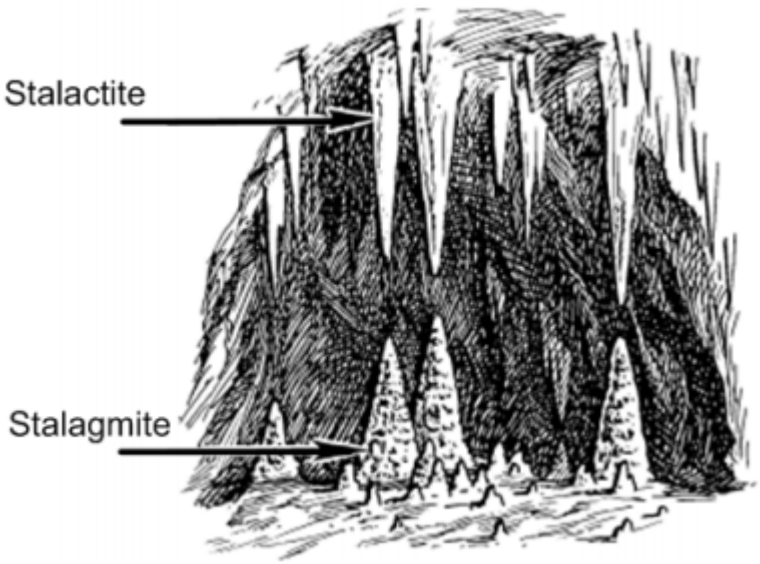
Multi-Stage: Preprocess

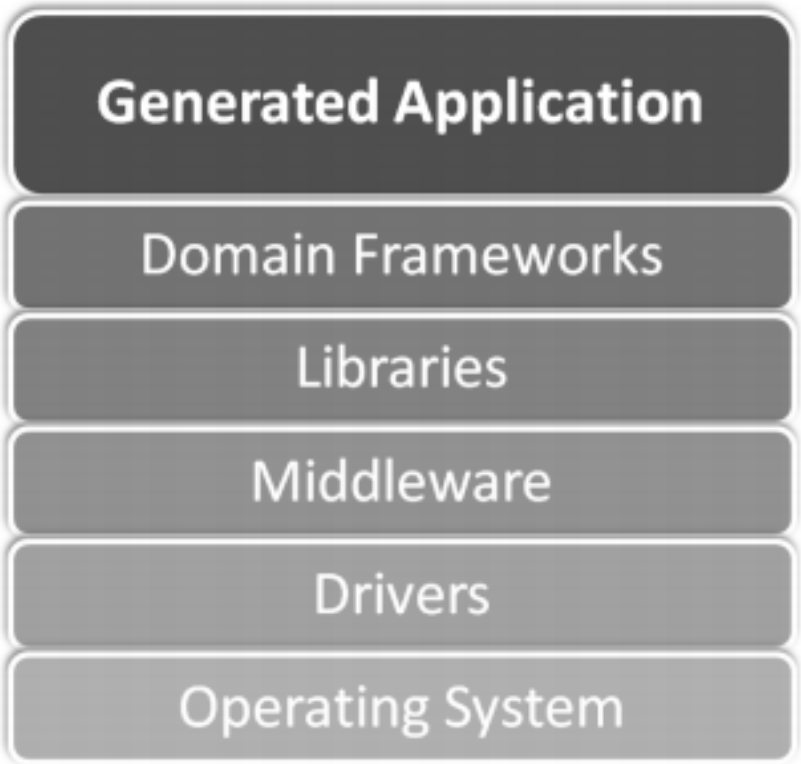


Adding an optional, modular emergency stop feature

Composite Blocks Transformation

Platform

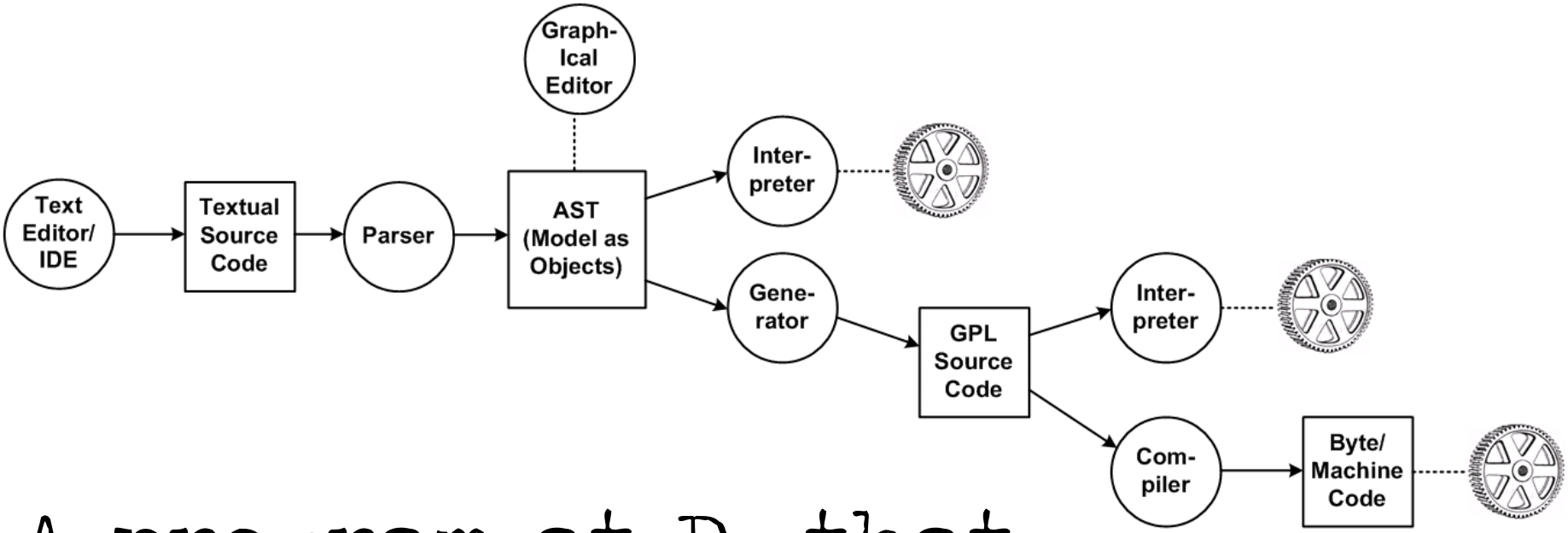

Stalactite
Stalagmite


Generated Application
Domain Frameworks
Libraries
Middleware
Drivers
Operating System

mbeddr

No Platform

Interpretation



A program at D_0 that acts on the structure of an input program at $D_{>0}$

Transformation

+ Code Inspection
+ Debugging
+ Performance & Optimization
+ Platform Conformance

Interpretation

mbeddr

Essentially Everything :-)

Transformation

+ Code Inspection
+ Debugging
+ Performance & Optimization
+ Platform Conformance

Interpretation

+ Turnaround Time
+ Runtime Change

Business Rules in Requirements

Def: Semantics

… via mapping to lower level

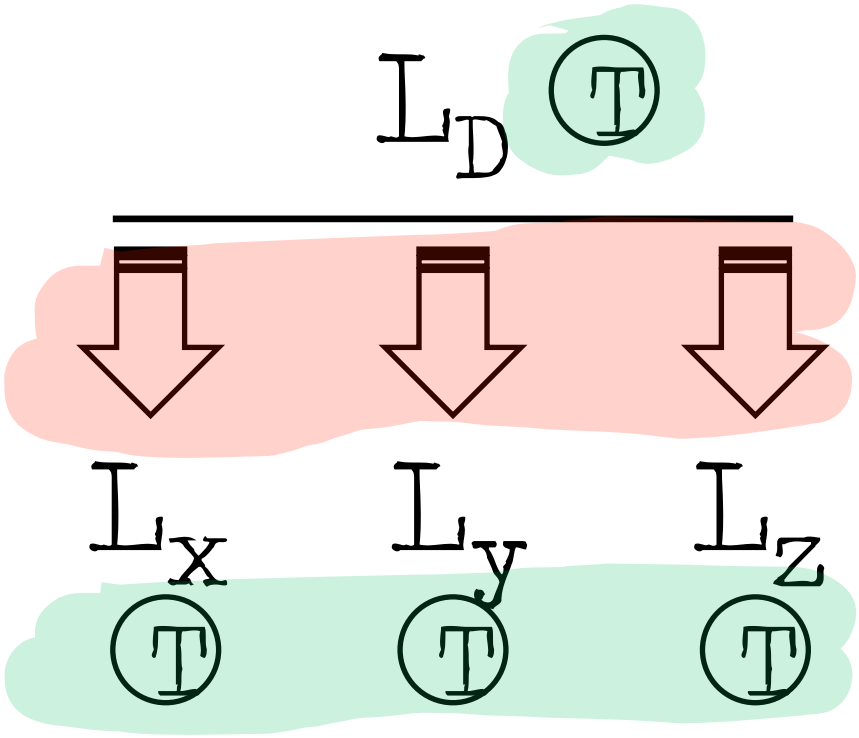
L_D

$\Downarrow$ Transformation
Interpretation

L_{D-1}

Multiple Mappings

… at the same time

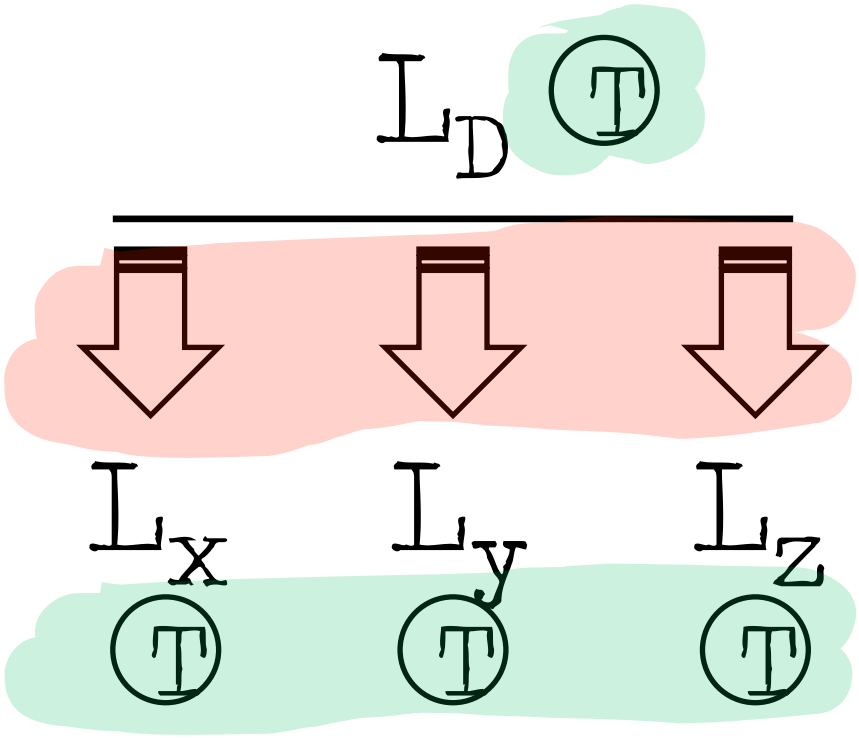


Similar Semantics?

all green!

Multiple Mappings

… at the same time

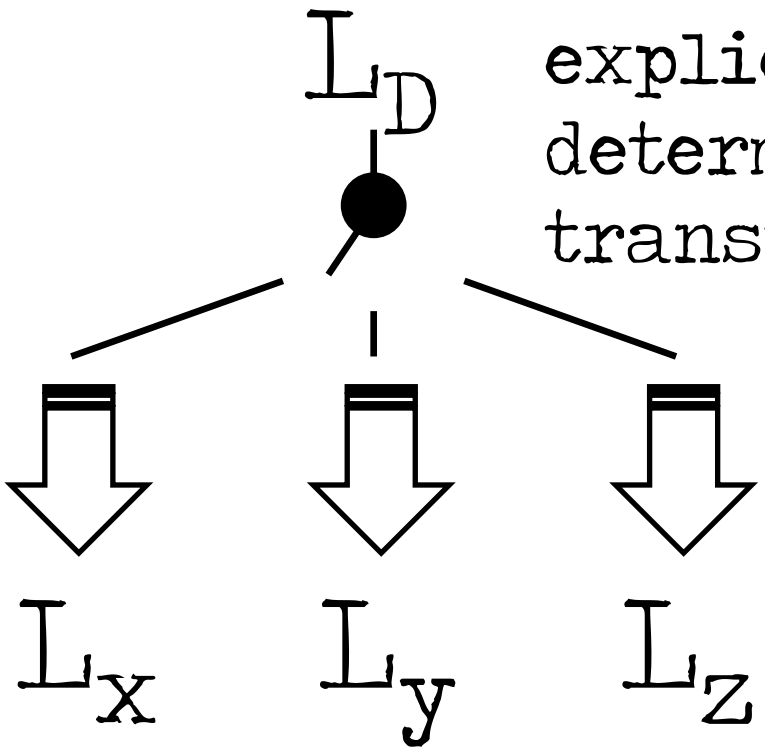


Similar Semantics?

all green!

Multiple Mappings

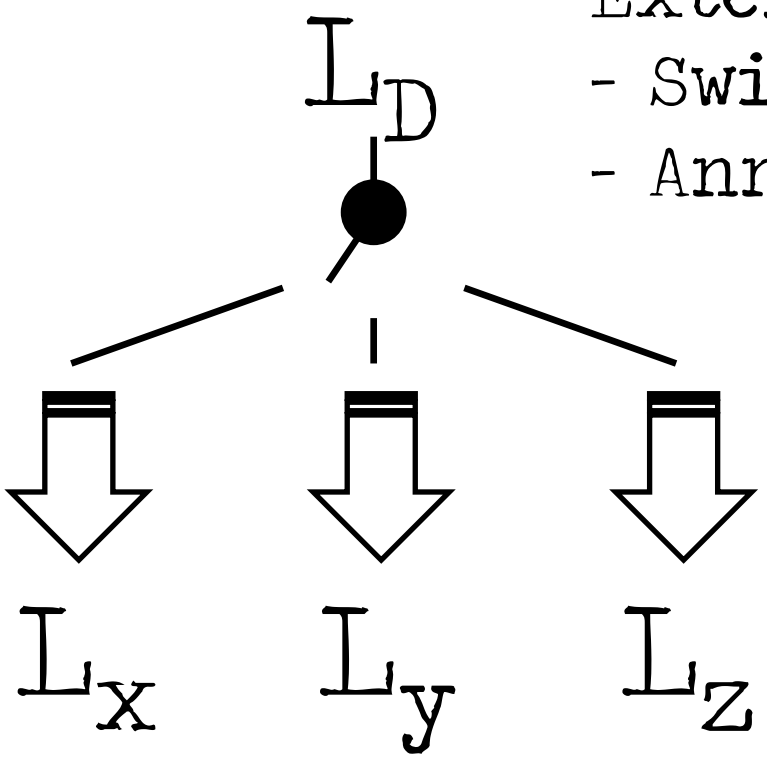
… alternatively, selectably



Extend L_D to include explicit data that determines transformation

Multiple Mappings

… alternatively, selectably



External Data:
- Switches
- Annotation Model

mbeddr

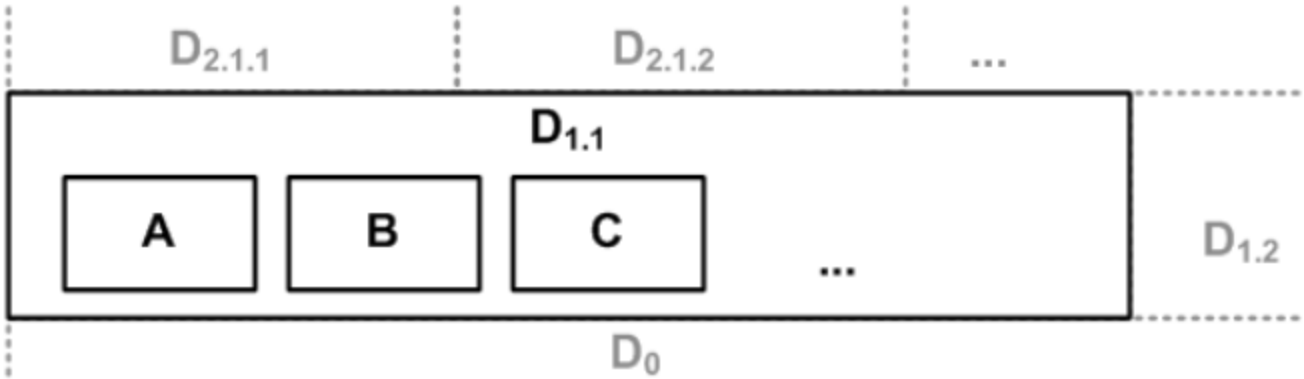
Build Config
Comp Static Wiring

6

Separation of Concerns

Several Concerns

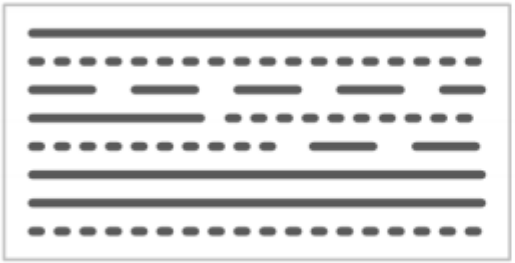
… in one domain



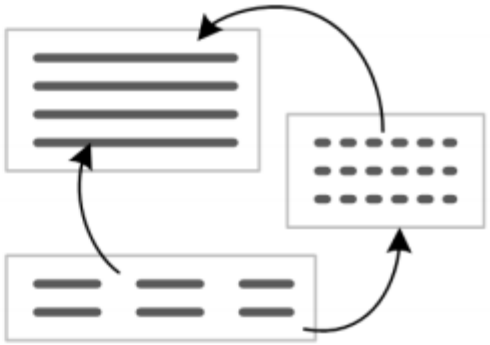
Several Concerns

… in one domain

integrated into
one fragment



separated into
several fragments



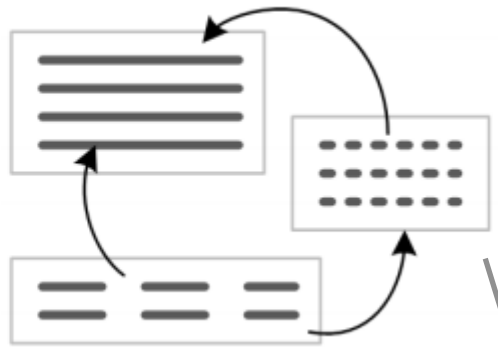
Components + Instances

Viewpoints

$$\forall r \in Refs_f \mid fo(r.to) = fo(r.from) = f$$

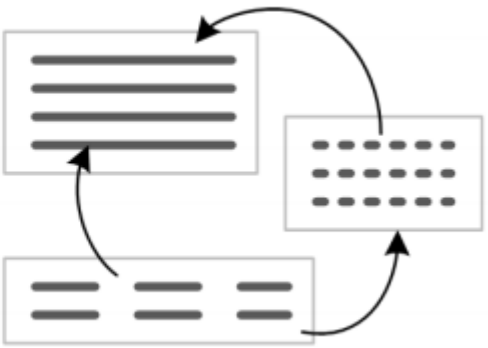
independent

$$\forall e \in E_f \mid lo(co(e)) = l$$



dependent

Viewpoints: Why?



Sufficiency

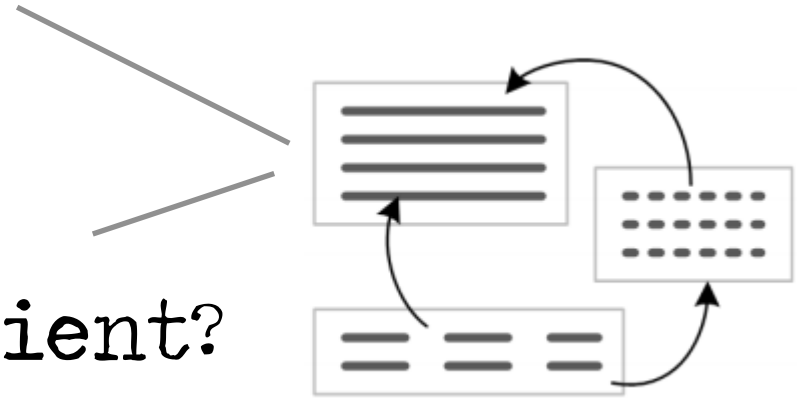
Different Stakeholders

Different Steps in Process - VCS unit!

Separate Requirements

Viewpoints

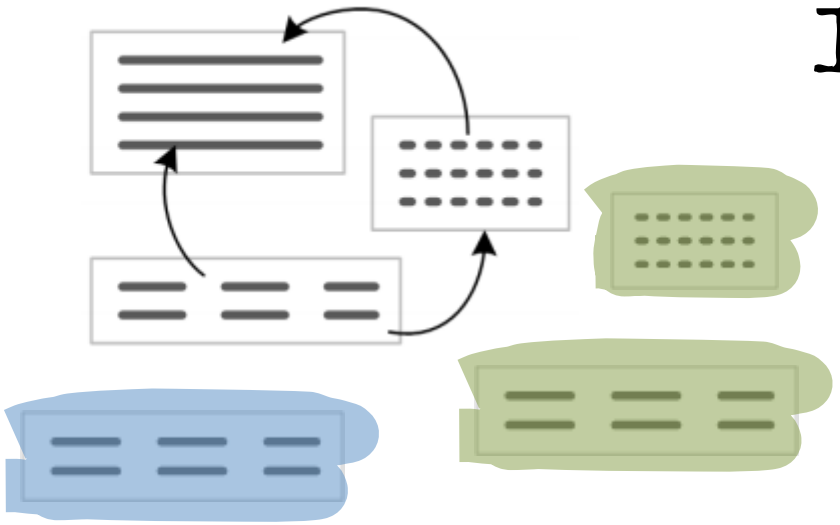
independent



sufficient?

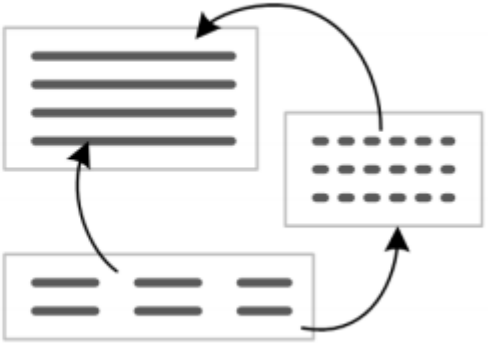
contains all the data for running a meaningful transformation

Viewpoints: Why?



1:n Relationships

Viewpoints



Well-defined Dependencies

No Cycles!

Avoid Synchronization!

(unless you use a projectional editor)

Sync in Comp + Interfaces

7

Completeness

Can you generate 100% of the code from the DSL program?

More generally: all of D_{-1}

Incomplete: What to do?

$$F_D$$

$$\Downarrow$$

$$OB(F_D) \mathrel{!=} F_{D-1}$$

Incomplete: What to do?

Manually written code!

$$F_D$$

$$\Downarrow$$

$$OB(F_D) == F_{D-1} + F_{D-1,\ man}$$

Manually written code?

Call "black box" code
(foreign functions)

State Machine Event Functions

Manually written code?

Call "black box" code
(foreign functions)

Embed L_{D-1} code in L_D program

All of mbeddr

Manually written code?

Call "black box" code
(foreign functions)

Embed L_{D-1} code in L_D program

Use composition mechanisms of
L_{D-1} (inheritance, patterns, aspects, ...)

Manually written code?

Call "black box" code (foreign functions)

Embed L_{D-1} code in L_D program

Use composition mechanisms of L_{D-1} (inheritance, patterns, aspects, ...)

Use protected regions (if you really have to...)

Manually written code?

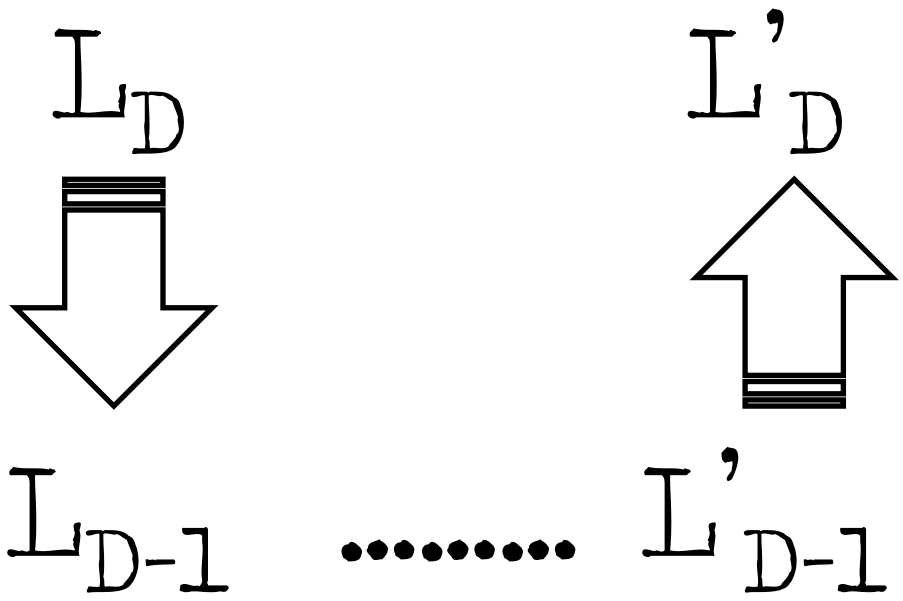
Call "black box" code (foreign functions)

Embed L_{D-1} code in L_D program

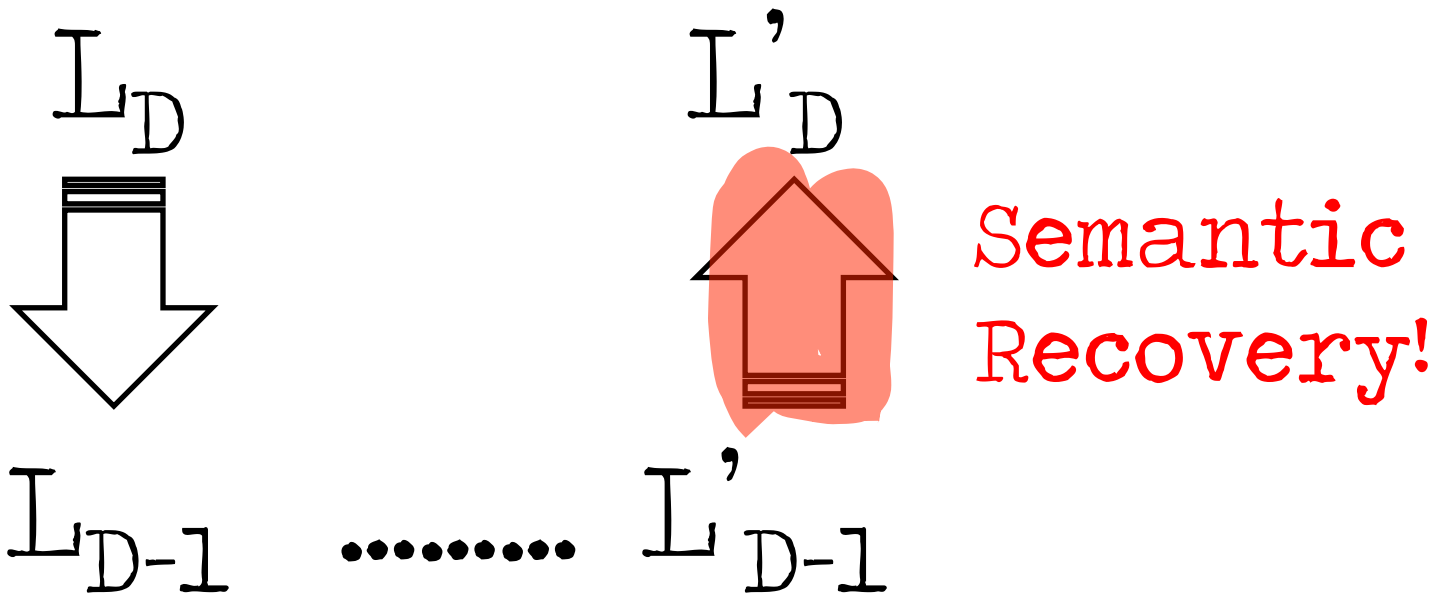
Use composition mechanisms of L_{D-1} (inheritance, patterns, aspects, ...)

Use protected regions (if you really have to...) DON'T!

Roundtripping



Roundtripping - Don't!



8

Fundamental Paradigms

Structure

Modularization, Visibility

Namespaces,
public/private
importing

Structure

Modularization, Visibility

- Namespaces,
- public/private
- importing
- ⇨ divide & conquer
- reuse
- stakeholder integration

mbeddr Chunks

Structure

Partitioning (Files)

VCS Unit
Unit of sharing
Unit of IP

!= logical modules

may influence language design

MPS models

Structure

Spec vs. Implementation

⇨ plug in different Impls
different stakeholders

Interfaces + Components

Structure

Specialization

Liskov substitution P
leaving holes ("abstract")

Structure

Specialization

Liskov substitution P
leaving holes ("abstract")

⇨ variants (in space)
evolution (over time)

Components Polymorphism

Behavior

Not all DSLs specify behavior

Some just **declare** behavior

This section is
not for those!

Behavior

Imperative

sequence of statements
changes program state

write	understand	debug	analyze	performance
simple	simple -	simple (step)	hard	good

C

Behavior

Functional

functions call other functions.
no state. No aliasing.

write	understand	debug	analyze	performance
simple -	simple	simple (tree)	good	good -

ACCENT Blocks

Behavior

Functional

pure expressions are
a subset of functional
(operators hard-wired)

guards
preconditions
derived attributes

Business Rules – Debugging

Behavior

Declarative

only facts and goals.
no control flow.
eval engine, solver (several)

write	understand	debug	analyze	performance
simple	simple -	hard	depends	often bad

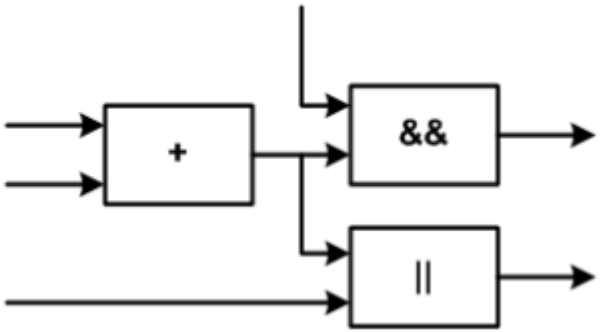
Behavior

Declarative

- concurrency
- constraint programming
- solving
- logic programming

Typing Rules

Behavior

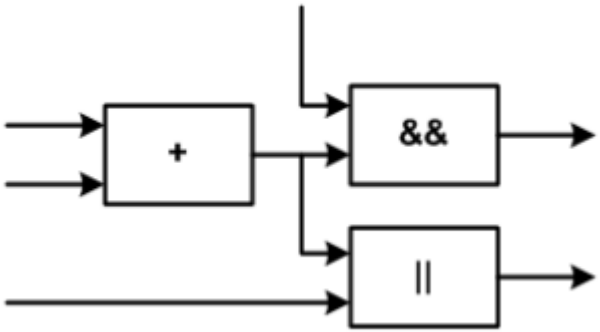


Data Flow

chained blocks consume continuous data that flows from block to block

write	understand	debug	analyze	performance
simple -	simple/hard	hard	simple	can be good

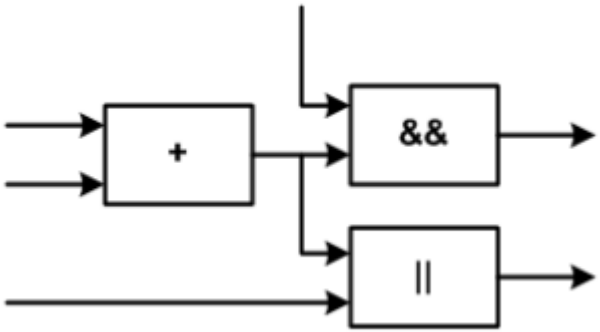
Behavior



Data Flow

continuous, calc on change
quantized, calc on new data
time triggered, calc every x

Behavior

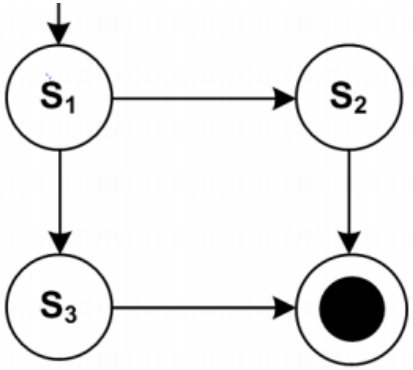


Data Flow

Embedded Programming
Enterprise ETL & CEP

ACCENT Blocks

Behavior



State Based

states, transitions,
guards, reactions

event driven, timed

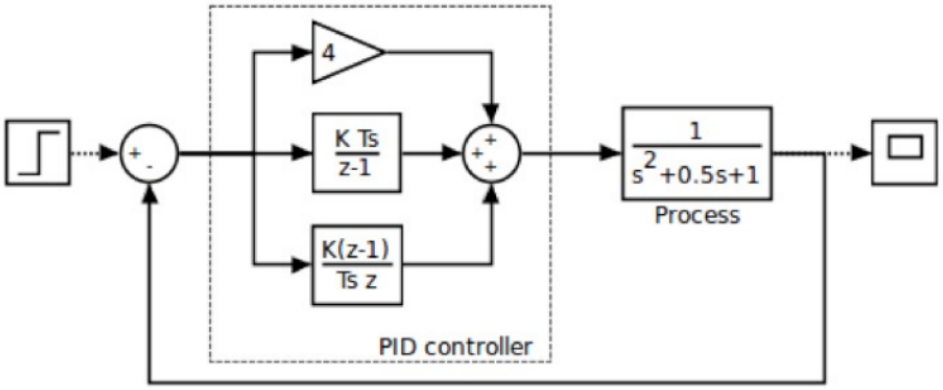
write	understand	debug	analyze	performance
simple -	simple/hard	s/h	simple +	can be good

State Machines

Behavior

Combinations

data flow uses functional, imperative or declarative lang inside block




```
synchronous blockType org::eclipselabs::damos::library::base::_discrete::DiscreteDerivative

input u
output y

parameter initialCondition = 0
parameter gain = 1(s) // normalized

behavior {

        stateful func main<initialCondition, gain, fs>(u) -> y {
                check<0, 1(s), 1(1/s)>(real) -> real

                static assert u is real() :
                        error "Input value must be numeric"

                static assert initialCondition is real() :
                        error "Initial condition must be numeric"

                static assert initialCondition is real() && u is real() => unit(initialCondition) == unit(u) :
                        error "Initial condition and input value must have same unit"

                static assert gain is real() :
                        error "Gain value must be numeric"

                eq u{-1} = initialCondition
                eq y{n} = fs * gain * (u{n} - u{n-1})
        }

}
```

Behavior

Combinations

- state machines use expressions in guards and often an imperative lang in actions

9

Modularity

Language Modularity, Composition and Reuse (LMR&C)

increase efficiency of DSL development

4 ways of composition:

Referencing

Reuse

Extension

Reuse

Language Modularity, Composition and Reuse (LMR&C)

increase efficiency of DSL development

4 **ways of composition:**

distinguished regarding **dependencies** and **fragment structure**

Dependencies:

do we have to know about the reuse when designing the languages?

Fragment Structure:

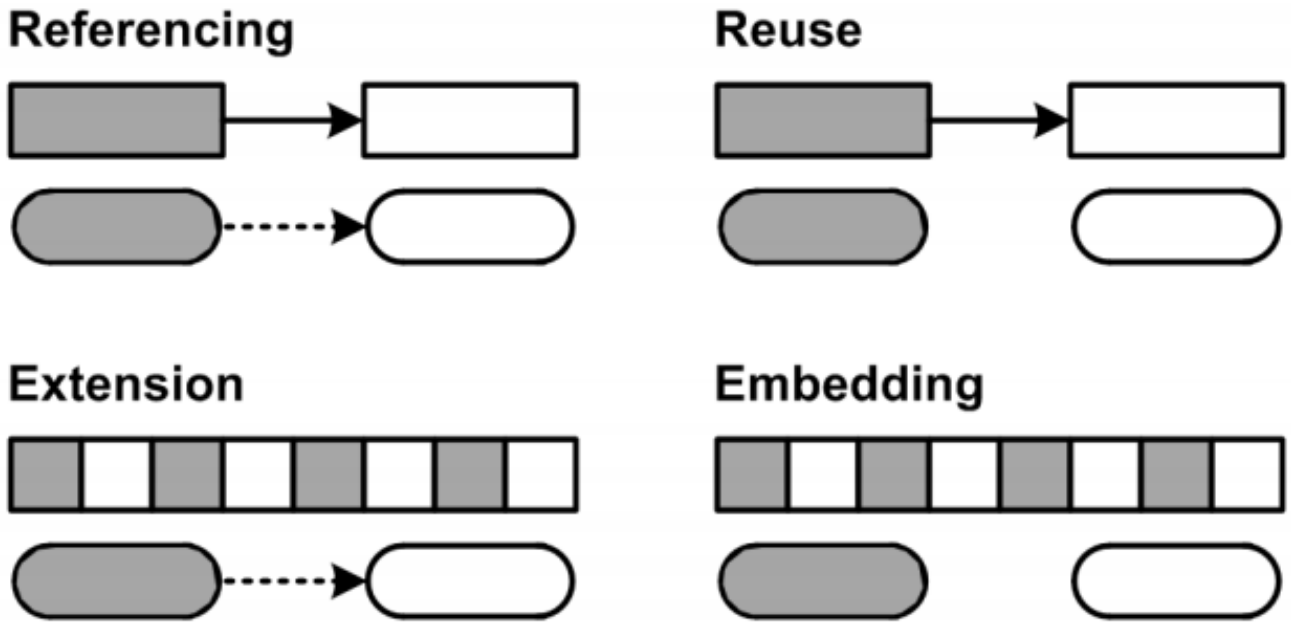
homogeneous vs. heterogeneous („mixing languages")

Dependencies & Fragment Structure:

languages dependencies	homogeneous	heterogeneous
independent	**Reuse**	**Embedding**
dependent	**Referencing**	**Extension**

fragment structure

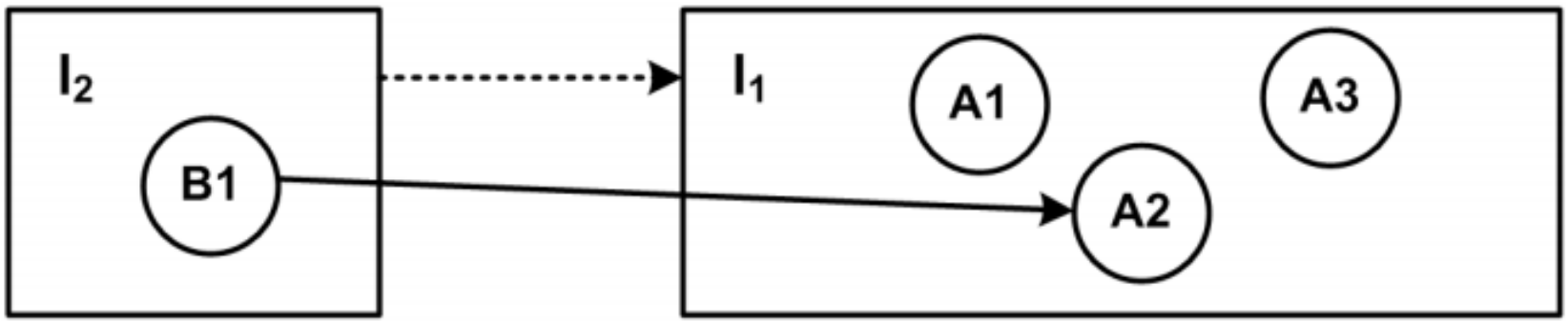
Dependencies & Fragment Structure:



Referencing

languages dependencies	homogeneous	heterogeneous
independent	**Reuse**	**Embedding**
dependent	**Referencing**	**Extension**

fragment structure

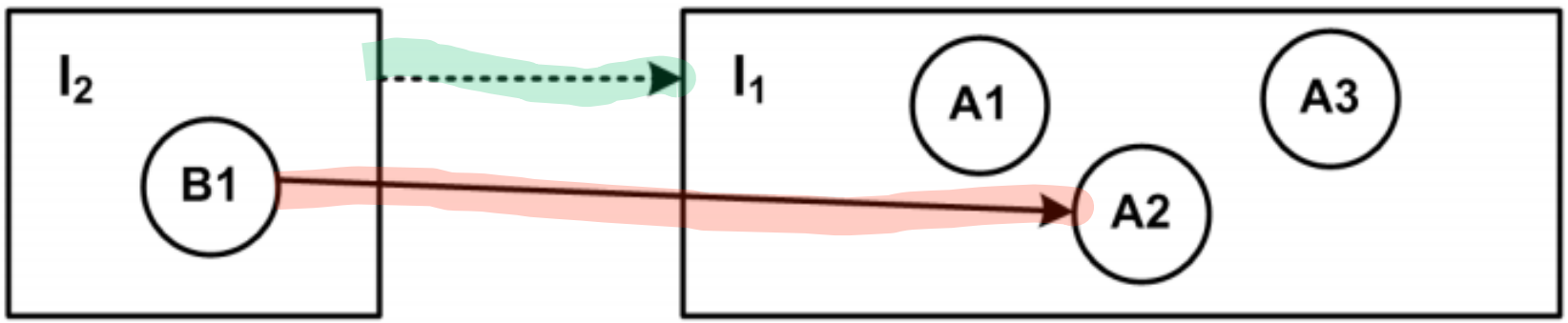


Referencing

languages dependencies	homogeneous	heterogeneous
independent	Reuse	Embedding
dependent	Referencing	Extension

fragment structure

Dependent



No containment

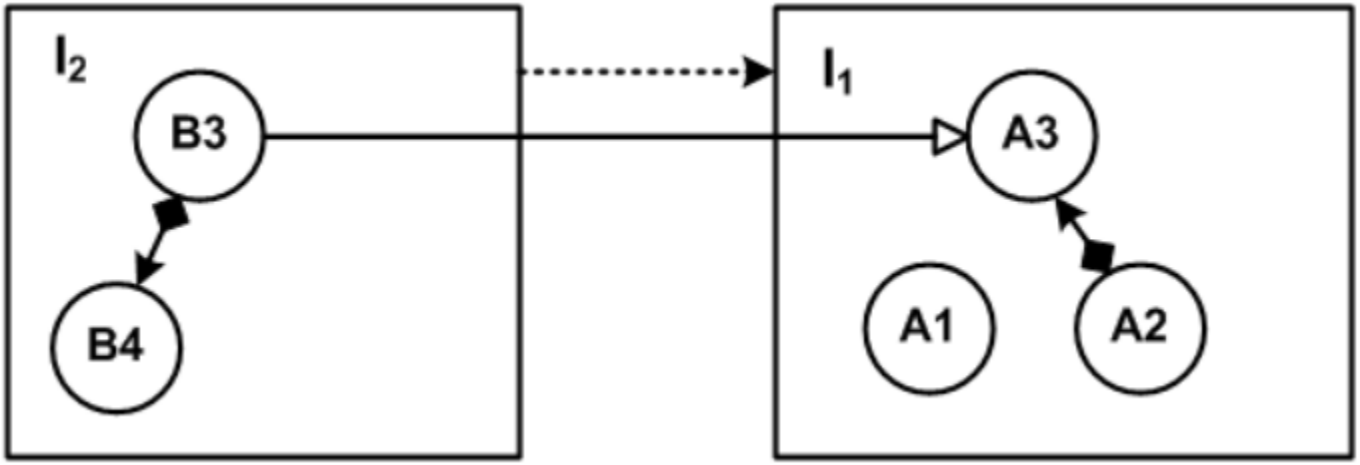
Referencing

Used in
Viewpoints

Extension

languages dependencies	homogeneous	heterogeneous
independent	**Reuse**	**Embedding**
dependent	**Referencing**	**Extension**

fragment structure

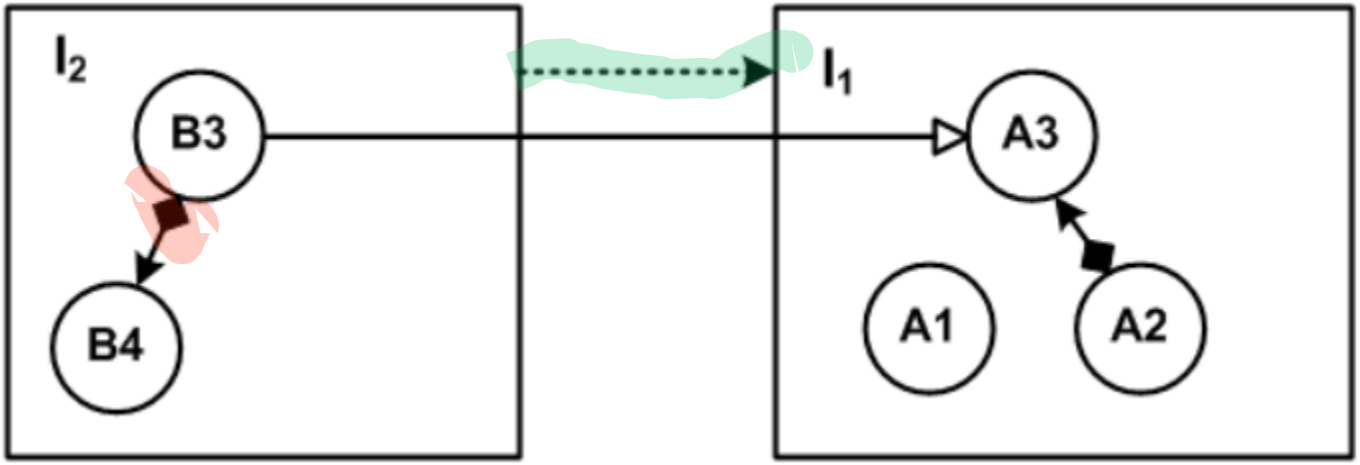


Extension

languages dependencies	homogeneous	heterogeneous
independent	**Reuse**	**Embedding**
dependent	**Referencing**	**Extension**

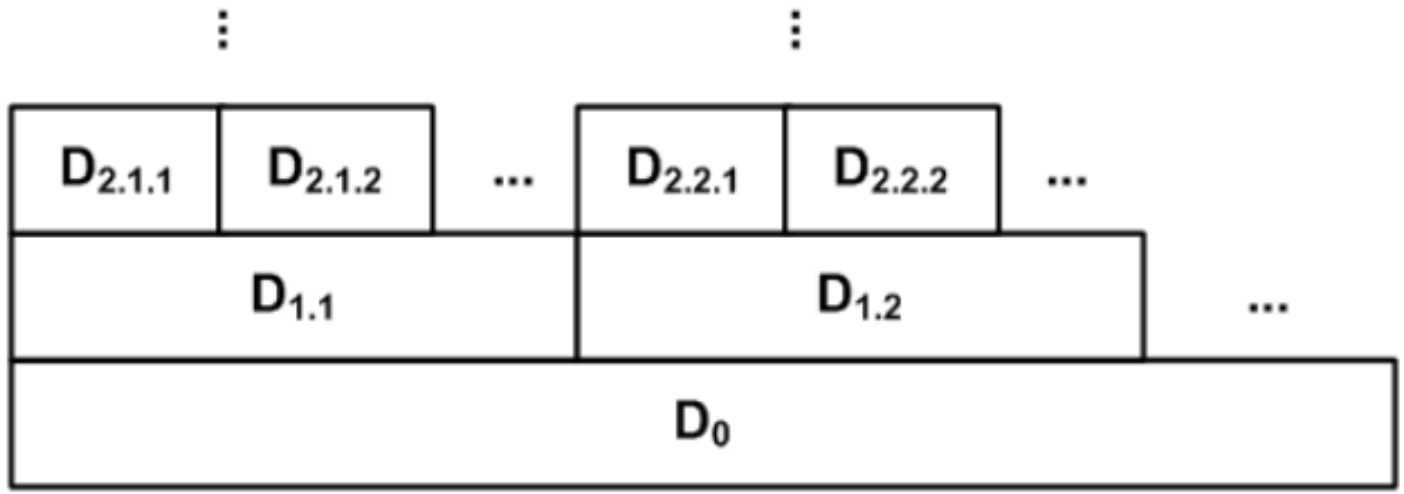
fragment structure

Dependent



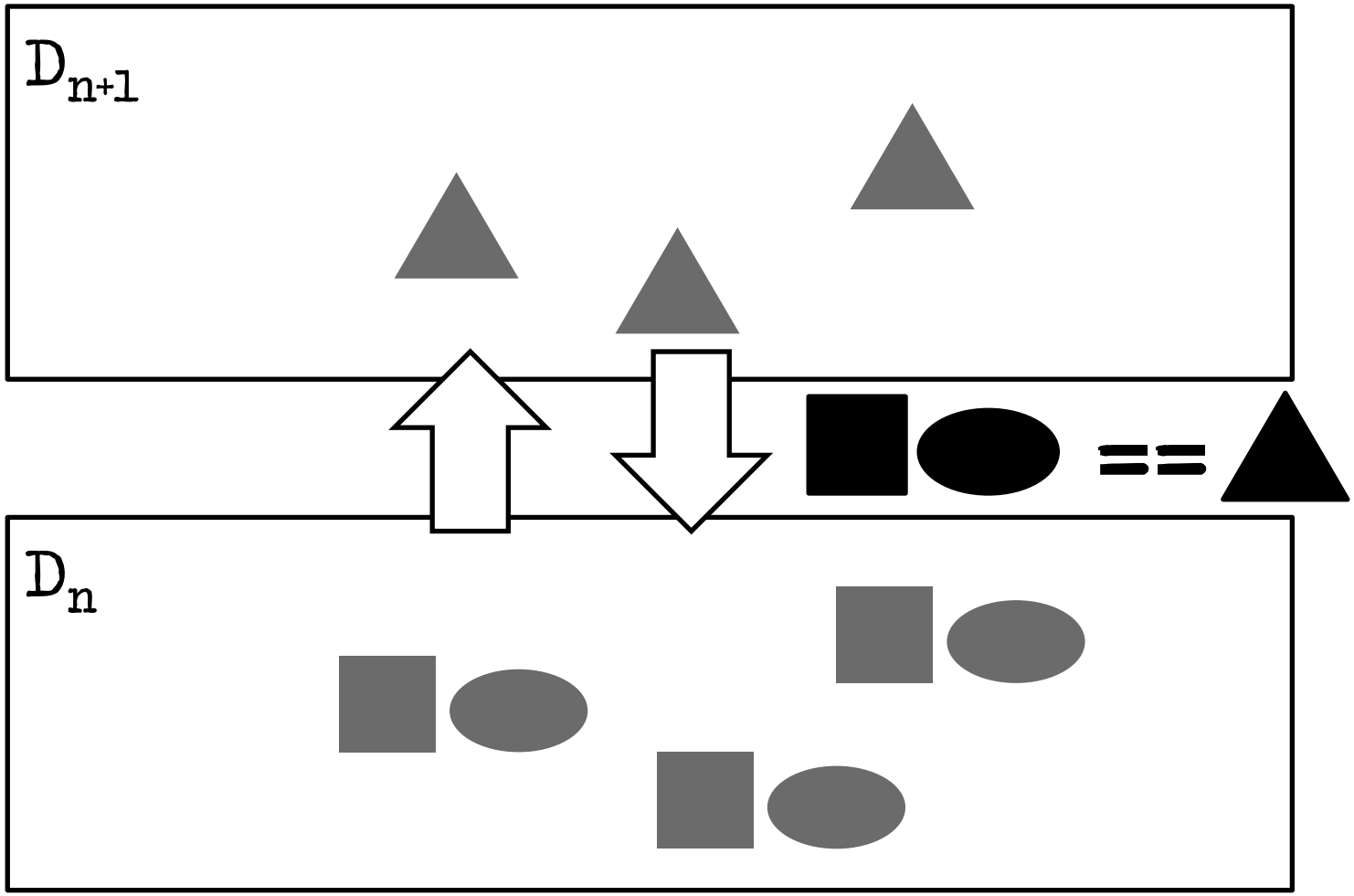
Containment

Extension

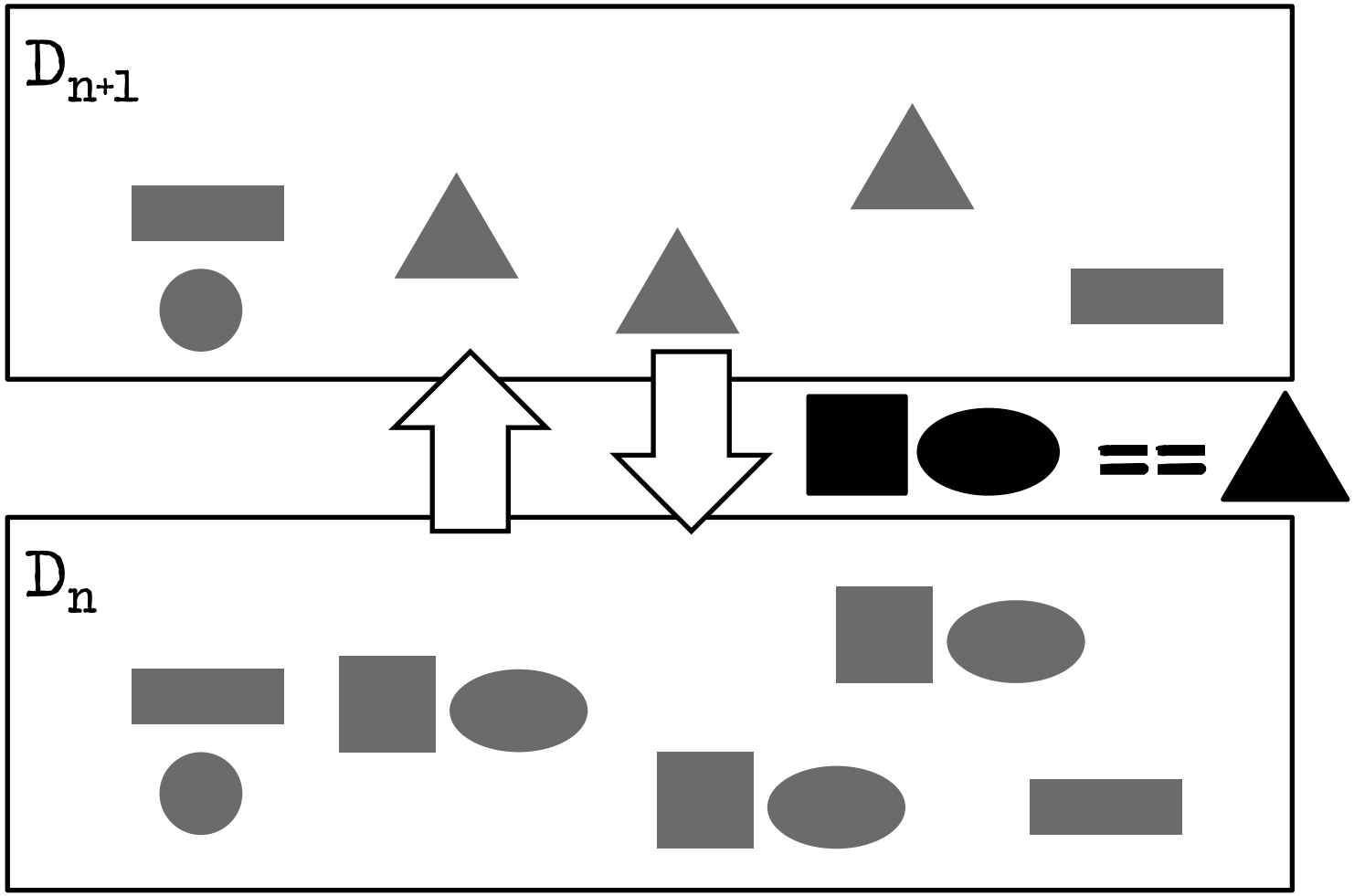


more specialized domains
more specialized languages

Extension

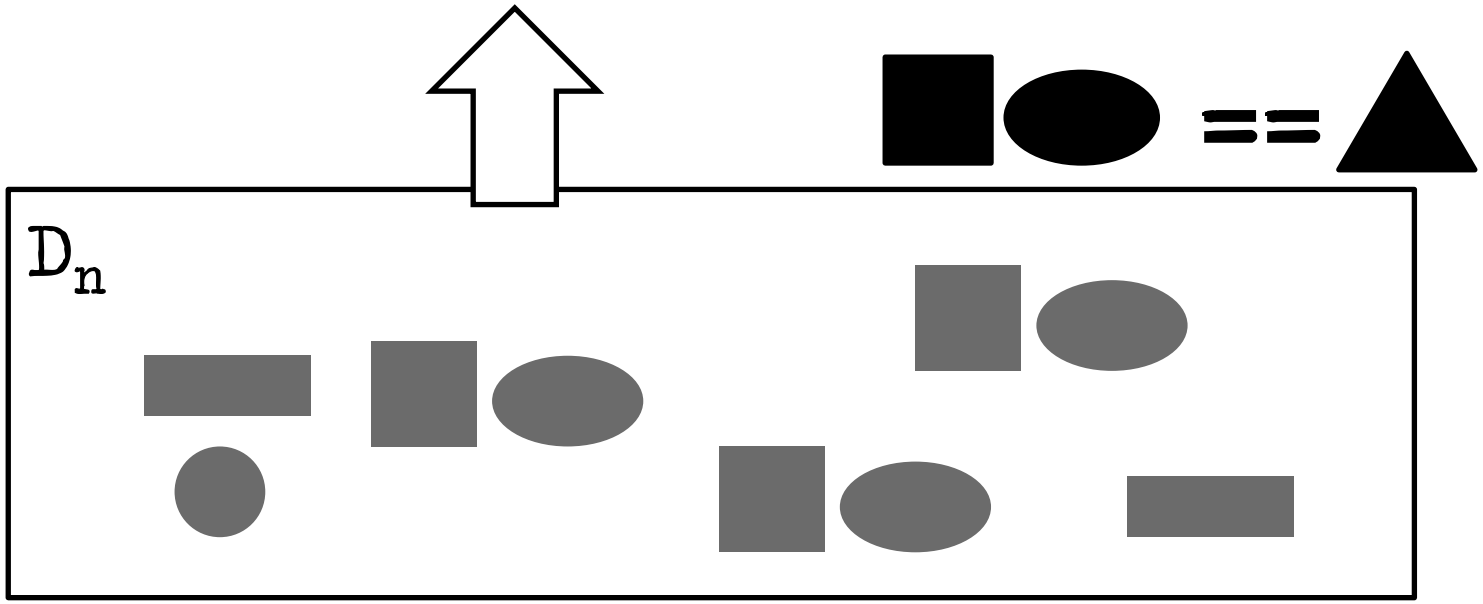


Extension



Extension

Good for **bottom-up** (inductive) domains, and for use by **technical** DSLs (people)



Extension

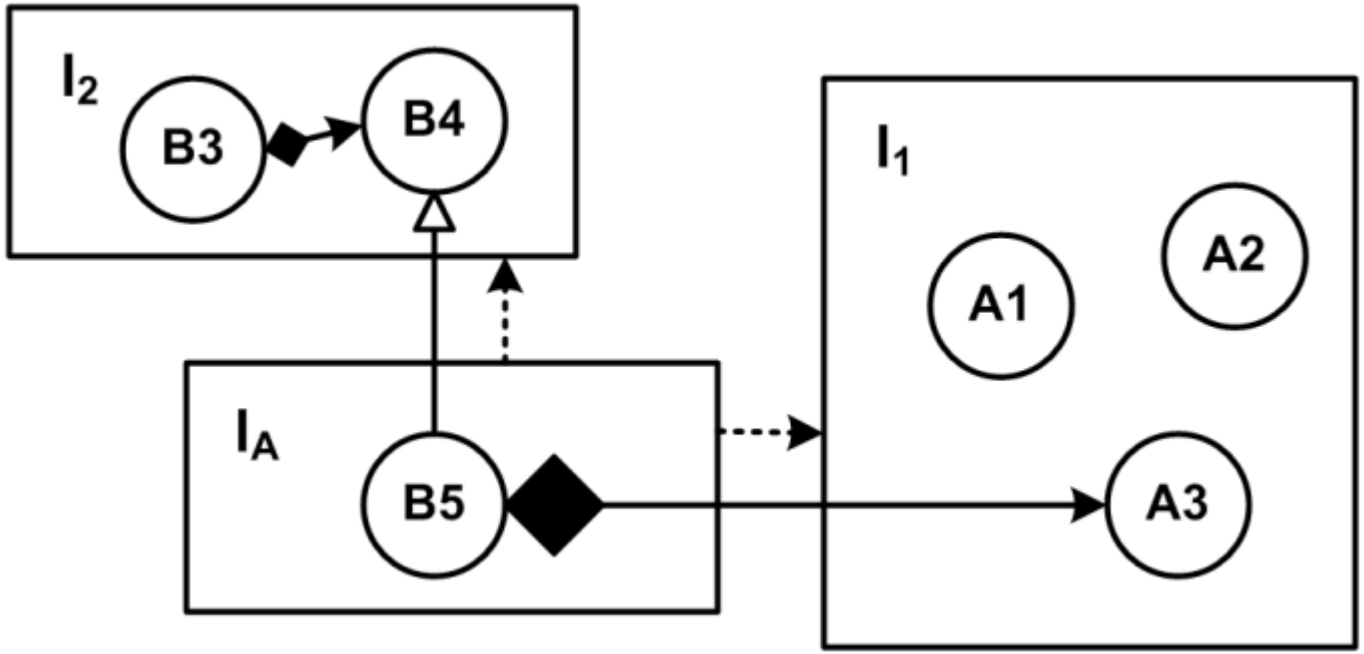
Drawbacks

- tightly bound to base
- potentially hard to analyze the combined program

Embedding

languages dependencies	homogeneous	heterogeneous
independent	**Reuse**	**Embedding**
dependent	**Referencing**	**Extension**

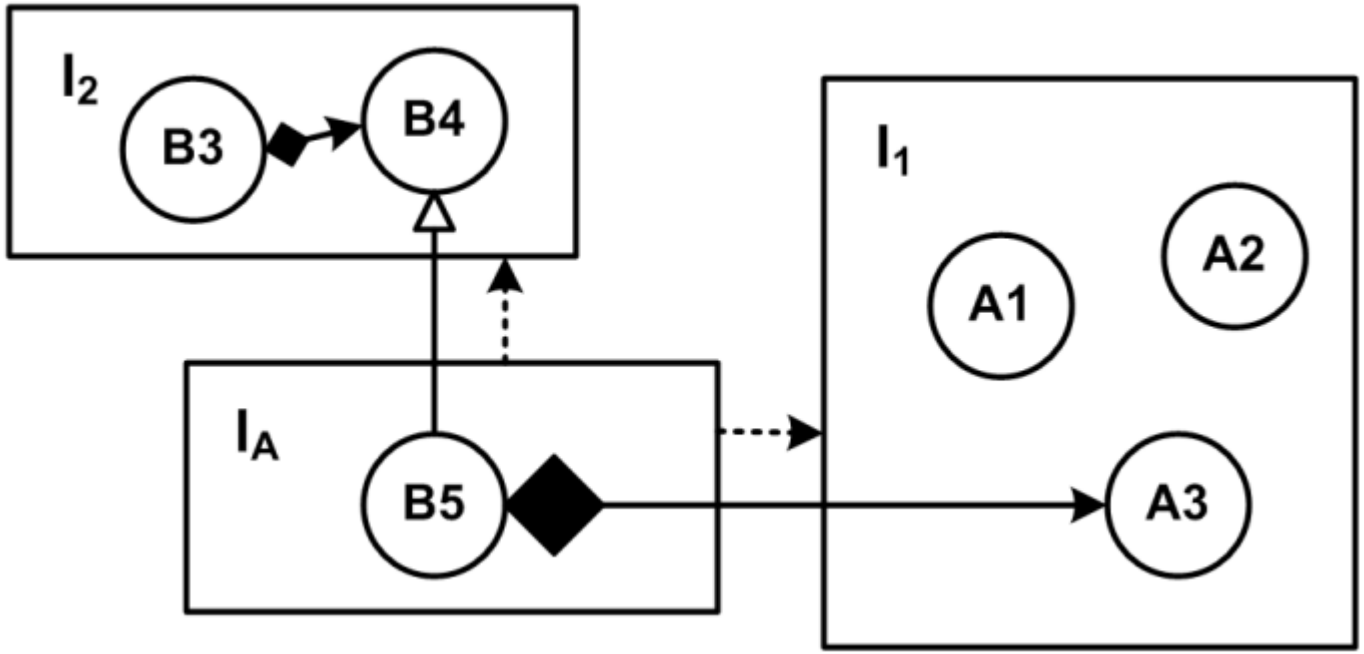
fragment structure



Embedding

languages dependencies	homogeneous	heterogeneous
independent	**Reuse**	**Embedding**
dependent	**Referencing**	**Extension**

fragment structure

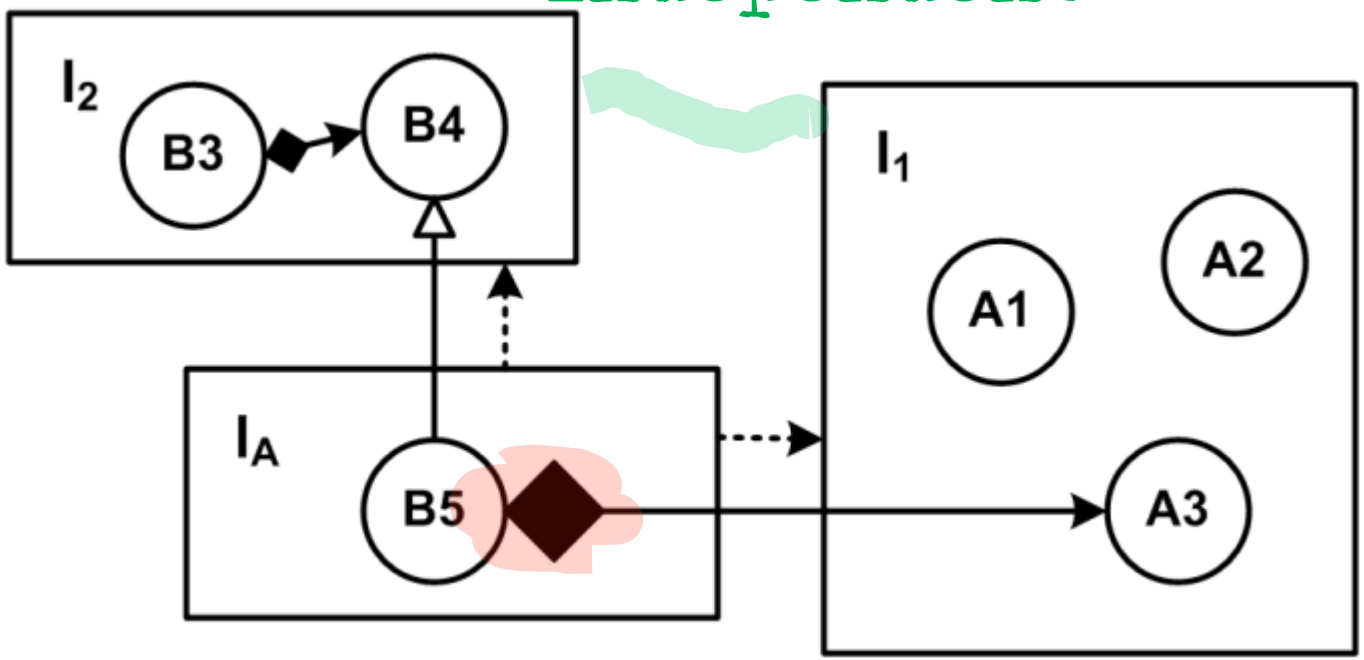


Embedding

languages dependencies	homogeneous	heterogeneous
independent	Reuse	Embedding
dependent	Referencing	Extension

fragment structure

Independent



Containment

Units in State Machines

Thank you!

voelter@acm.org
www.voelter.de
@markusvoelter