Language Workbenches
Opportunities and Challenges for V&V

Markus Völter

**Bernd Kolb, Dan Ratiu, Zaur Molotnikov,
Domenik Pavletic, Kolja Dumman,
Sascha Lisson, Tamas Szabo, Niko Stotz,**

voelter { ingenieurbüro für softwaretechnologie // itemis

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

0 Languages, Models, Programs

1 Language Workbenches

2 JetBrains MPS

3 mbeddr

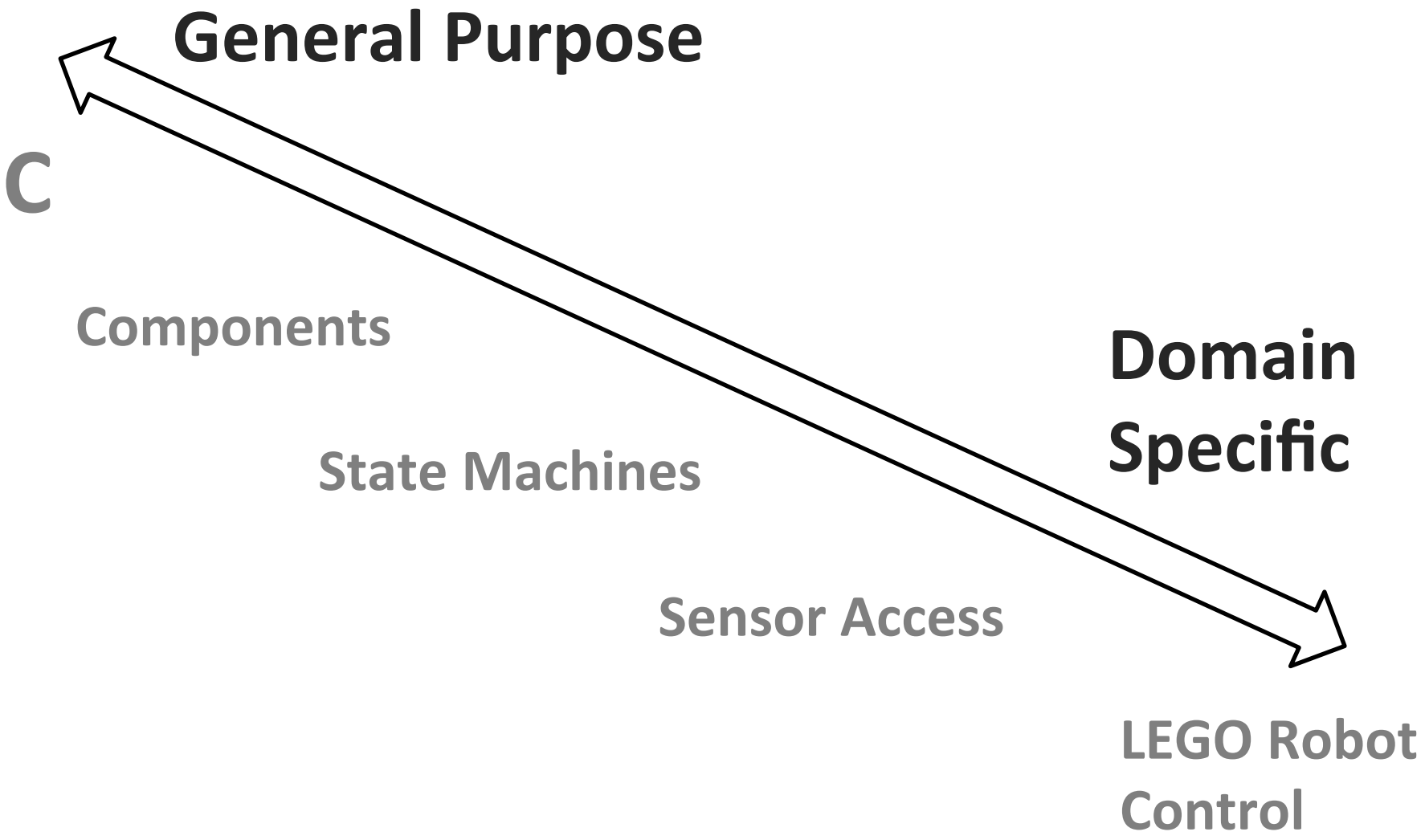
4 V&V in mbeddr

5 Opportunities & Challenges

0

Languages, Models, Programs

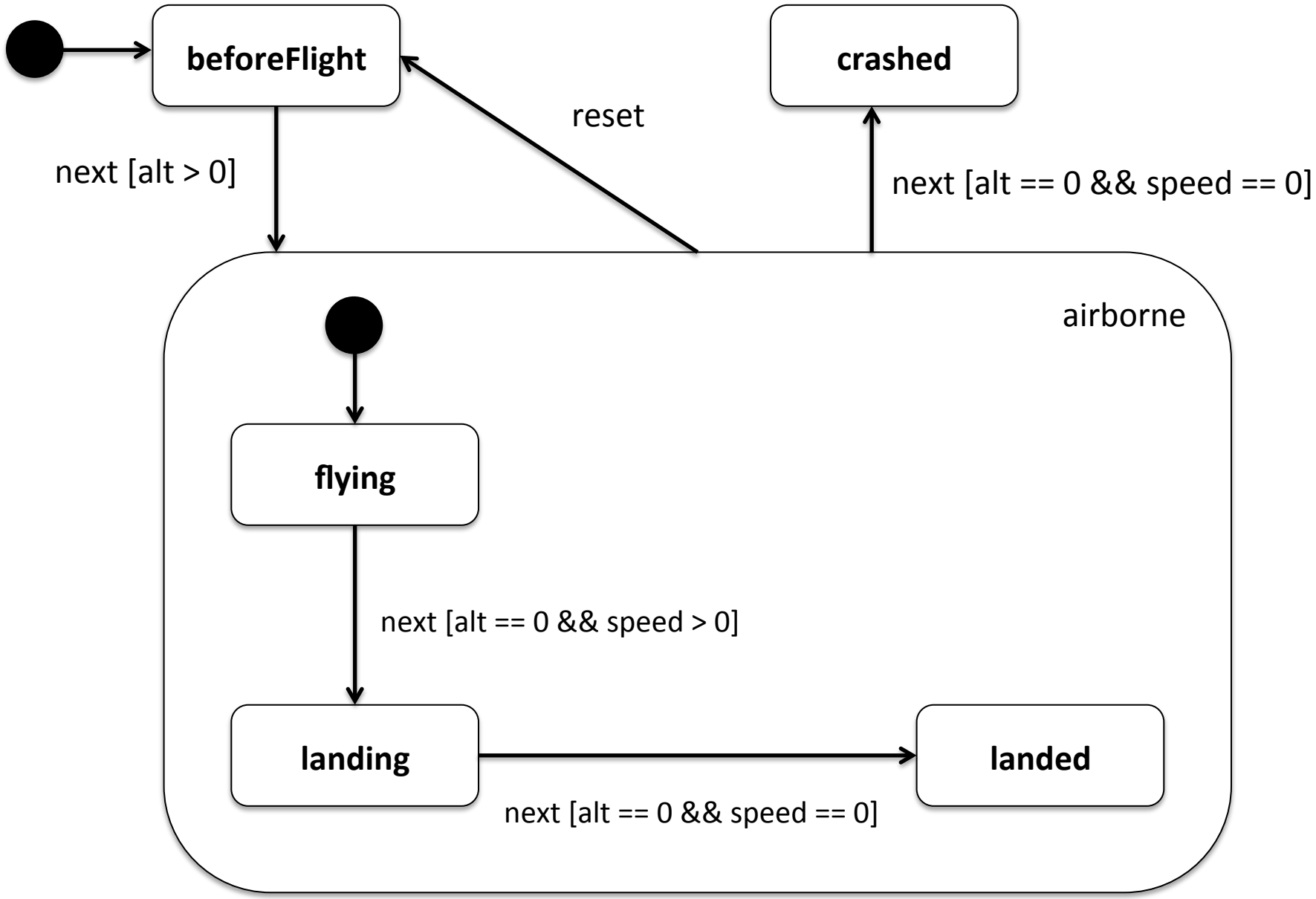
	more in GPLs	more in DSL
Domain Size	large and complex	smaller and well-defined
Designed by	guru or committee	a few engineers and domain experts
Language Size	large	small
Turing-completeness	almost always	often not
User Community	large, anonymous and widespread	small, accessible and local
In-language 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
State Machines
Sensor Access
Domain Specific
LEGO Robot Control

Model or Code?

```
Trackpoint* makeTP(uint16 alt, int16 speed) {
  static int8 trackpointCounter = 0;
  trackpointCounter++;
  Trackpoint* tp = ((Trackpoint*) malloc(sizeof Trackpoint));
  tp->id = trackpointCounter;
  tp->timestamp = trackpointCounter;
  tp->alt = alt
  tp->speed = speed
  return tp;
}
```

Model or Code?



Model or Code?

```
statemachine HierarchicalFlightAnalyzer initial = beforeFlight {
  in next()
  in reset()
  out crashNotification() -> raiseAlarm
  state beforeFlight {
    on next [tp->alt > 0 m] -> airborne
  }
  composite state airborne initial = flying {
    on reset [ ] -> beforeFlight
    on next [tp->alt == 0 m && tp->speed == 0 mps] -> crashed
    state flying {
      on next [tp->alt == 0 m && tp->speed > 0 mps] -> landing
      on next [tp->speed > 200 mps] -> airborne
      on next [tp->speed > 100 mps] -> airborne
    }
    state landing {
      on next [tp->speed == 0 mps] -> landed
      on next [ ] -> landing
    }
    state landed {
    }
  }
  state crashed {
  }
}
```

Model or Code?

```
statemachine HierarchicalFlightAnalyzer initial = beforeFlight {
  in next(Trackpoint* tp)
  in reset()
  out crashNotification() -> raiseAlarm
  readable var int16 points = 0
  state beforeFlight {
    on next [tp->alt > 0 m] -> airborne
    exit { points += TAKEOFF; }
  }
  composite state airborne initial = flying {
    on reset [ ] -> beforeFlight { points = 0; }
    on next [tp->alt == 0 m && tp->speed == 0 mps] -> crashed
    state flying {
      on next [tp->alt == 0 m && tp->speed > 0 mps] -> landing
      on next [tp->speed > 200 mps] -> airborne { points += VERY_HIGH_SPEED; }
      on next [tp->speed > 100 mps] -> airborne { points += HIGH_SPEED; }
    }
    state landing {
      on next [tp->speed == 0 mps] -> landed
      on next [ ] -> landing { points--; }
    }
    state landed {
      entry { points += LANDING; }
    }
  }
  state crashed {
    entry { send crashNotification(); }
  }
}
```

Model or Code?

Does it really matter?

What is the difference?

Who cares?

We don‘t want to

model,

we want to

program!

We don't want to model, we want to program!

… at different levels of **abstraction**

… from different **viewpoints**

… **integrated!**

We don't want to model, we want to program!

… with different degrees of **domain-specificity**

… with suitable **notations**

… **with suitable expressiveness**

We don‘t want to

model,

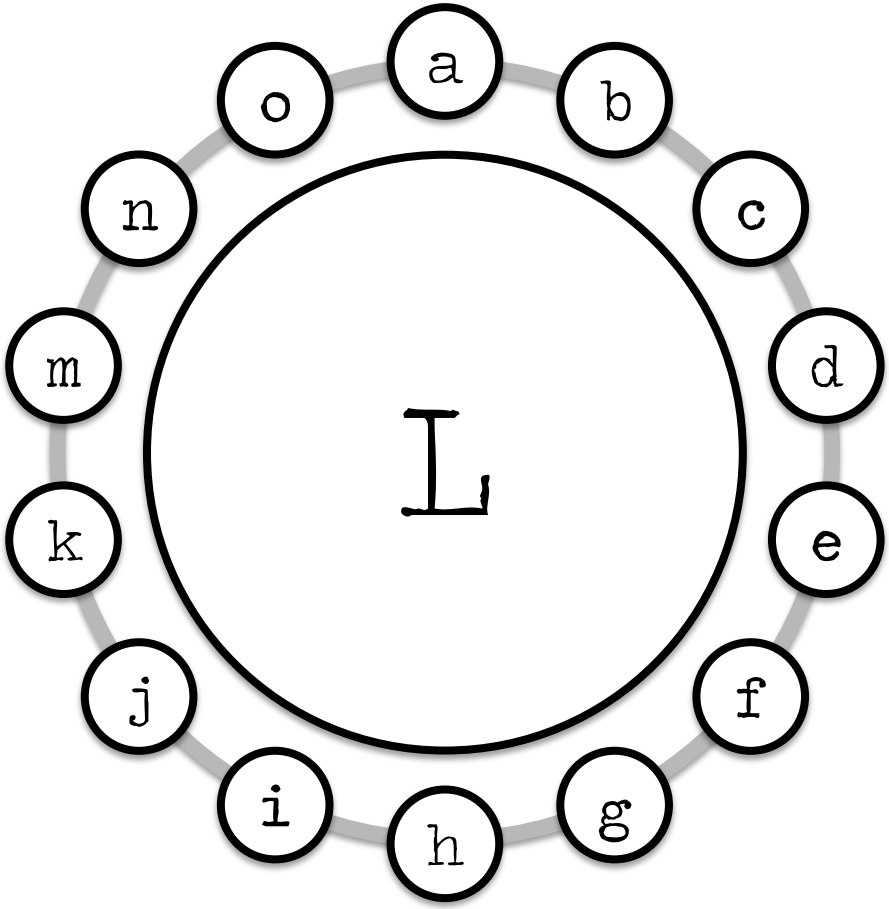
we want to

program!

and always:

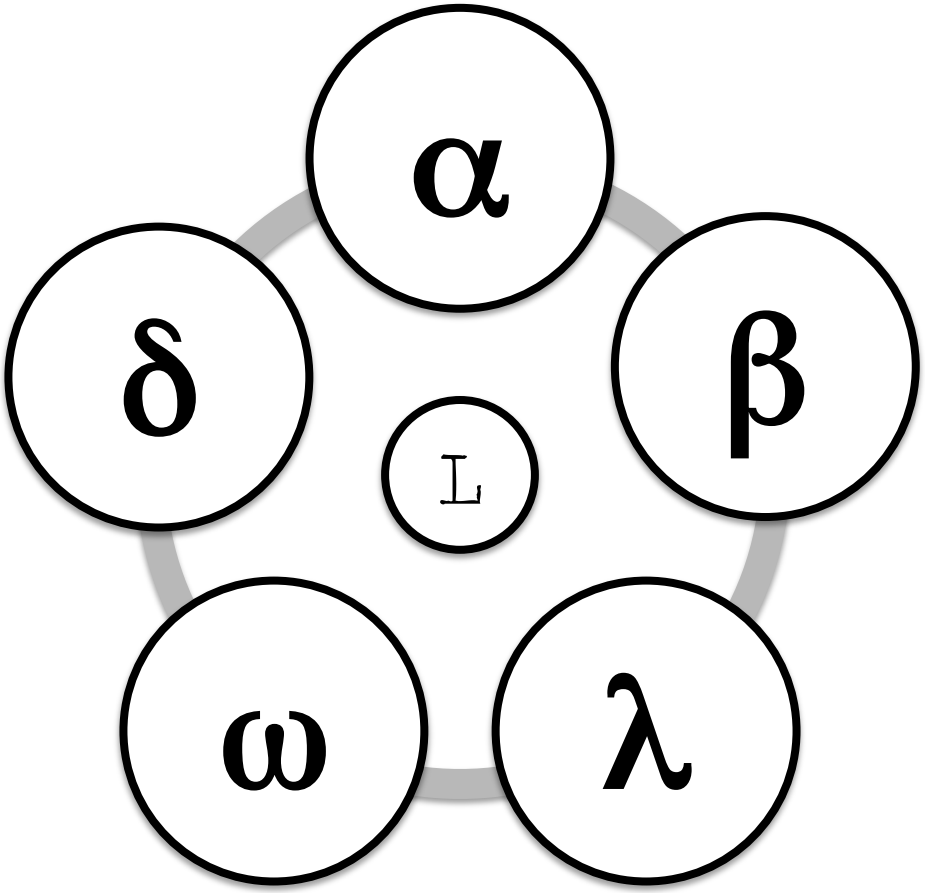
precise and tool processable

Big Language



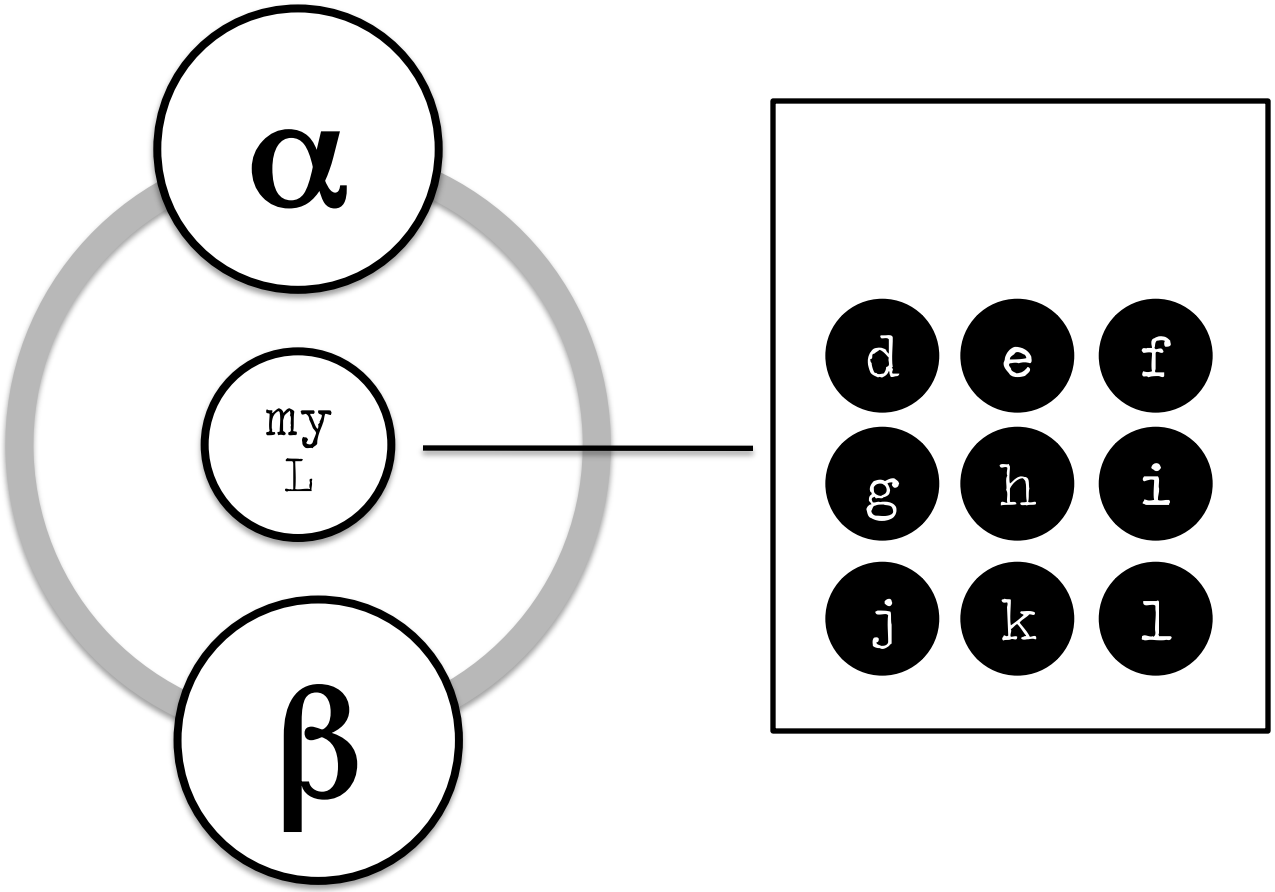
with **many** first class concepts!

Small Language



with a **few, orthogonal** and **poweful** concepts

Modular Language



with many **optional**,
composable modules

1st

Language Workbenches

Language Workbench

(Martin Fowler)

Language Workbench

(Martin Fowler)

Freely
define
languages and
integrate
them

Language Workbench

(Martin Fowler)

use
persistent
abstract
representation

?

Language Workbench

(Martin Fowler)

language ::=

schema

+ editors

+ generators

Language Workbench

(Martin Fowler)

projectional editing ?

Language Workbench

(Martin Fowler)

persist

incomplete

or

contradictory

information

Language Workbench

(Martin Fowler)

powerful

editing

testing

refactoring

debugging

groupware

language definition

implies

IDE definition

Language Workbench

(Martin Fowler)

support for

„classical“ **programming**

+

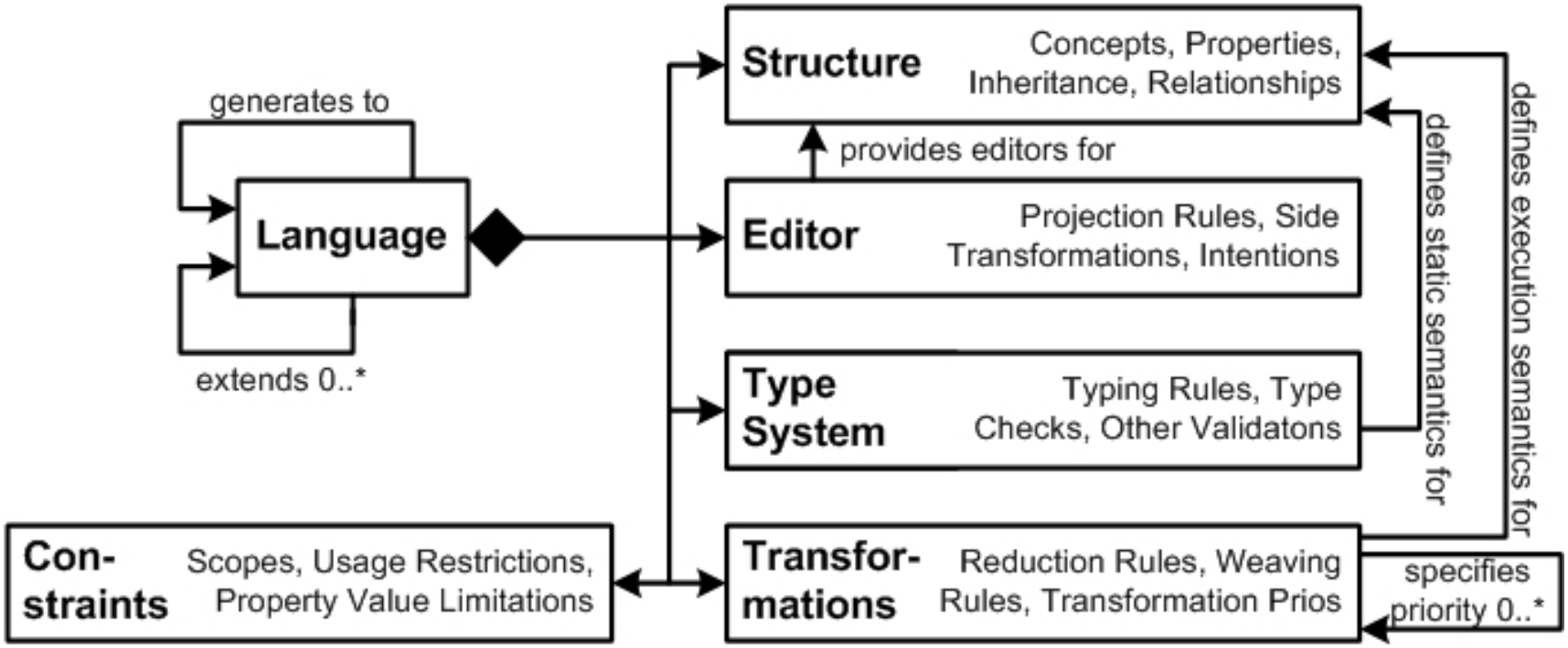
„classical“ and **modeling**

2

JetBrains MPS

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

[Language Workbench]

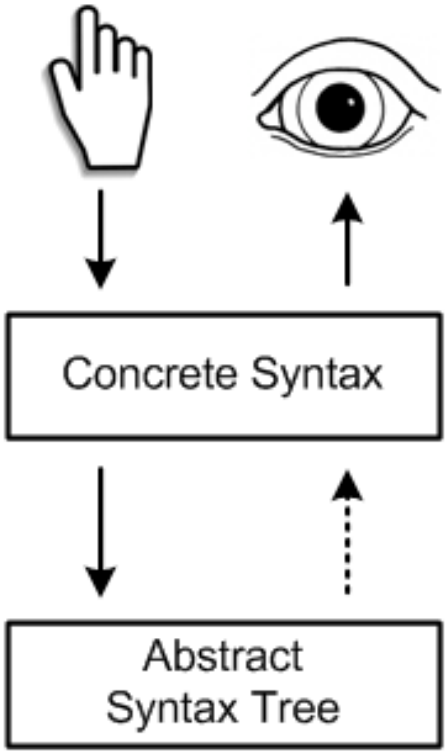


+ Refactorings, Find Usages, Syntax Coloring, Debugging, ...

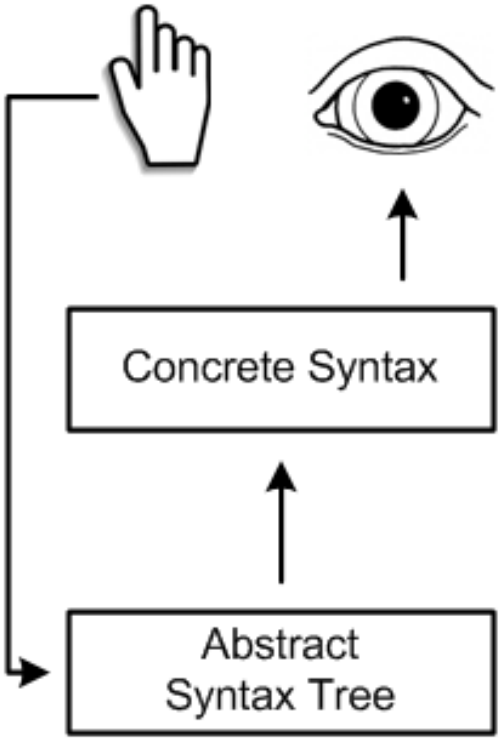
Projectional Editing

[Projectional Editing]

Parsing



Projectional Editing



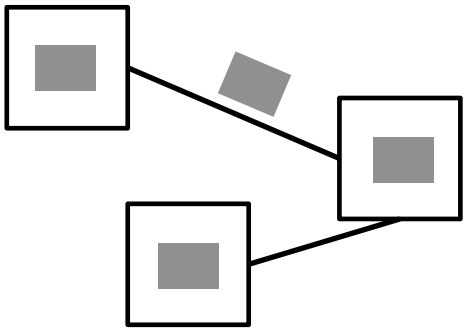
[Projectional Editing]
Syntactic Flexibility

Regular Code/Text

Mathematical

Tables

Graphical



[Projectional Editing]
Syntactic Flexibility

Regular Code/Text

```
//[ A documentation comment with references
  [ to @arg(data) and @arg(dataLen)
void aSummingFunction(int8[] data, int8 dataLen) {
  int16 sum;
  for (int8 i = 0; i < dataLen; i++) {
    sum += data[i];
  } for
} aSummingFunction (function)
```

Mathematical

```
double midnight2(int32 a, int32 b, int32 c) {
```

$$\text{return}\ \frac{-b + \sqrt{b^2 - \sum_{i=1}^{4} a * c}}{2 * a};$$

```
} midnight2 (function)
```

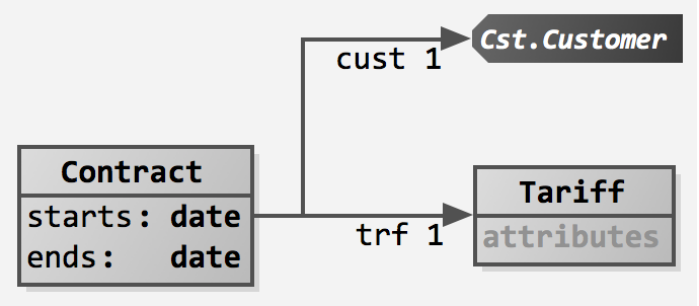
Tables

```
int16 decide(int8 spd, int8 alt) {
  return
```

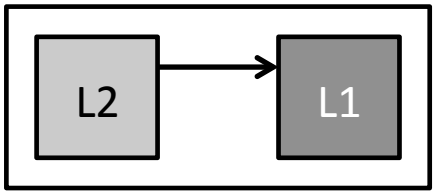
	spd > 0	spd > 100
alt < 0	1	1
alt == 0	10	20
alt > 0	30	40
alt > 100	50	60

```
otherwise 0;
} decide (function)
```

Graphical



[Projectional Editing]
Language Composition



Separate Files

Type System
Transformation
Constraints

In One File

Type System
Transformation
Constraints
Syntax
IDE

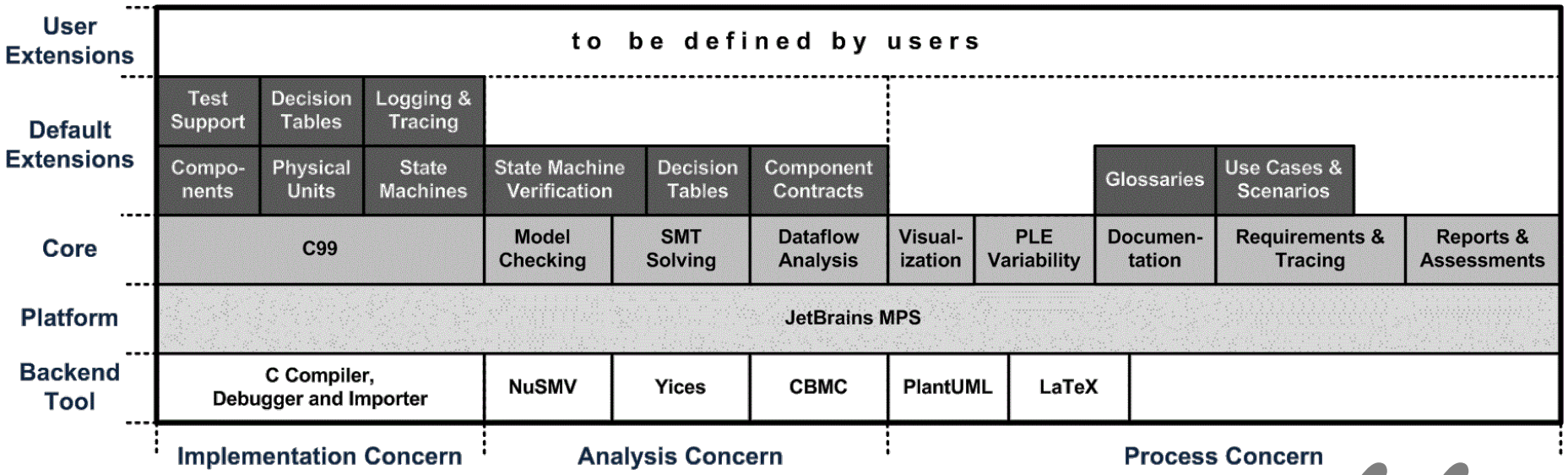
mbeddr

50+ extensions to C
10+ extensions to requirements lang.

3

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

fortiss itemis

BMW Car IT

Bundesministerium
für Bildung
und Forschung

Research Project 2011 - 2013

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

Commercial Use and Extension

itemis
SIEMENS

Research Platform

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

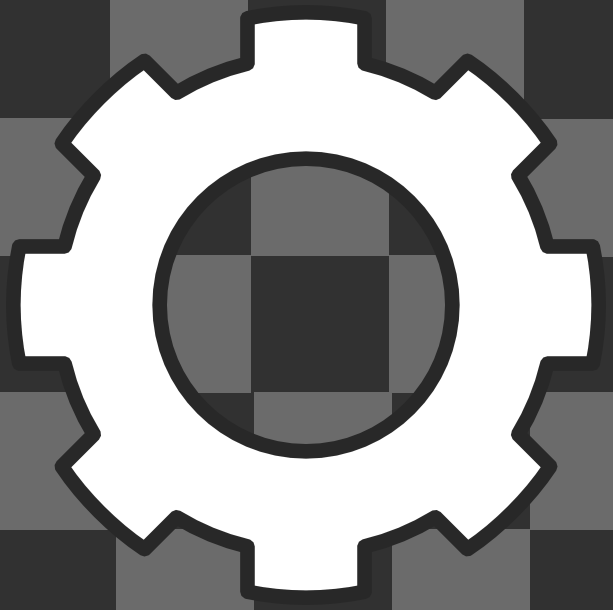
20+ Projects in various stages

by various “Big Name” companies.

Branching out into the Domains

Finance, Insurance, Requirements

DEMO



[Accidental Benefits]

Better Abstractions & Notations

Easier to Review and Validate

Better Modularity (Interf. / Comp.)

Improved Testability

Domain-Specific Testing Support

Makes Tests easier to Validate

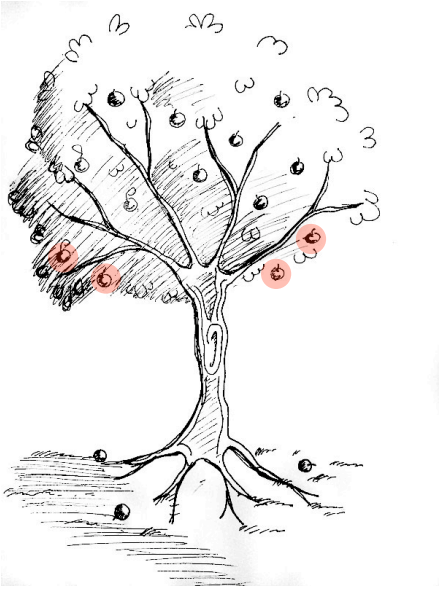
Domain-Specific Debugging

Abstractions are not Leaky!

4

V&V in mbeddr

[Type System]

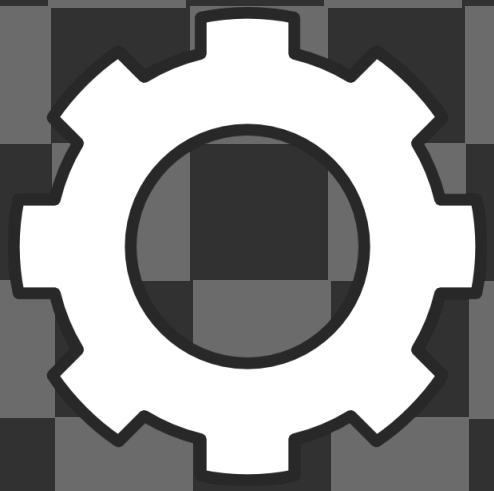


Unit Checking
Custom Constraints

```
void calcVerticalSpeed(TrackpointWithVertical* prev,
  if (prev == null) {
    cur->vSpeed = 0 mps;
  } else {
    int16
    int8/s/ dTime = cur->time - prev->time;
    cur->vSpeed = dTime / dAlt;
  } if
} calcVerticalSpeed (function)
```

Error: type int16 /s · m^(−1)/ is not a subtype of int16 /mps/

DEMO



Units; Layers

[DSCV] Molotnikov, Völter, Ratiu @ ASE 2014

Domain-Specific C Verification

Synergy between

Language Engineering & Verification

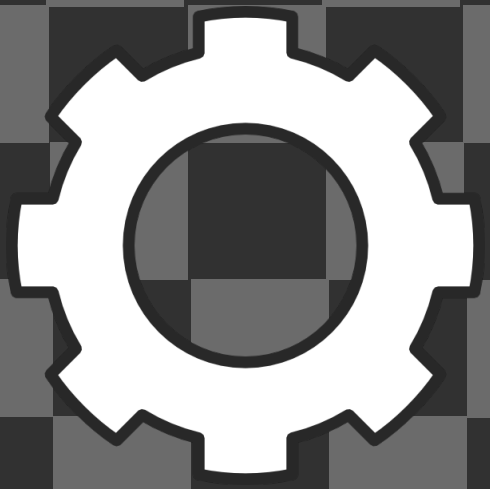
[DSCV] Molotnikov, Völter, Ratiu @ ASE 2014

Domain-Specific C Verification

Alternative 1

1. Make Program Expressive
2. Verify on Model Level
3. Report on Model Level

DEMO



PLE Verification

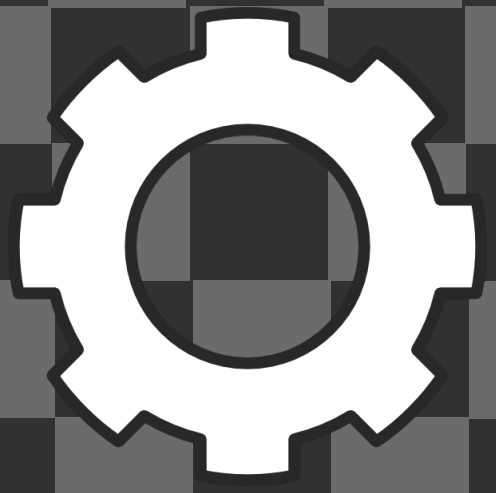
[DSCV] Molotnikov, Völter, Ratiu @ ASE 2014

Domain-Specific C Verification

Alternative 2

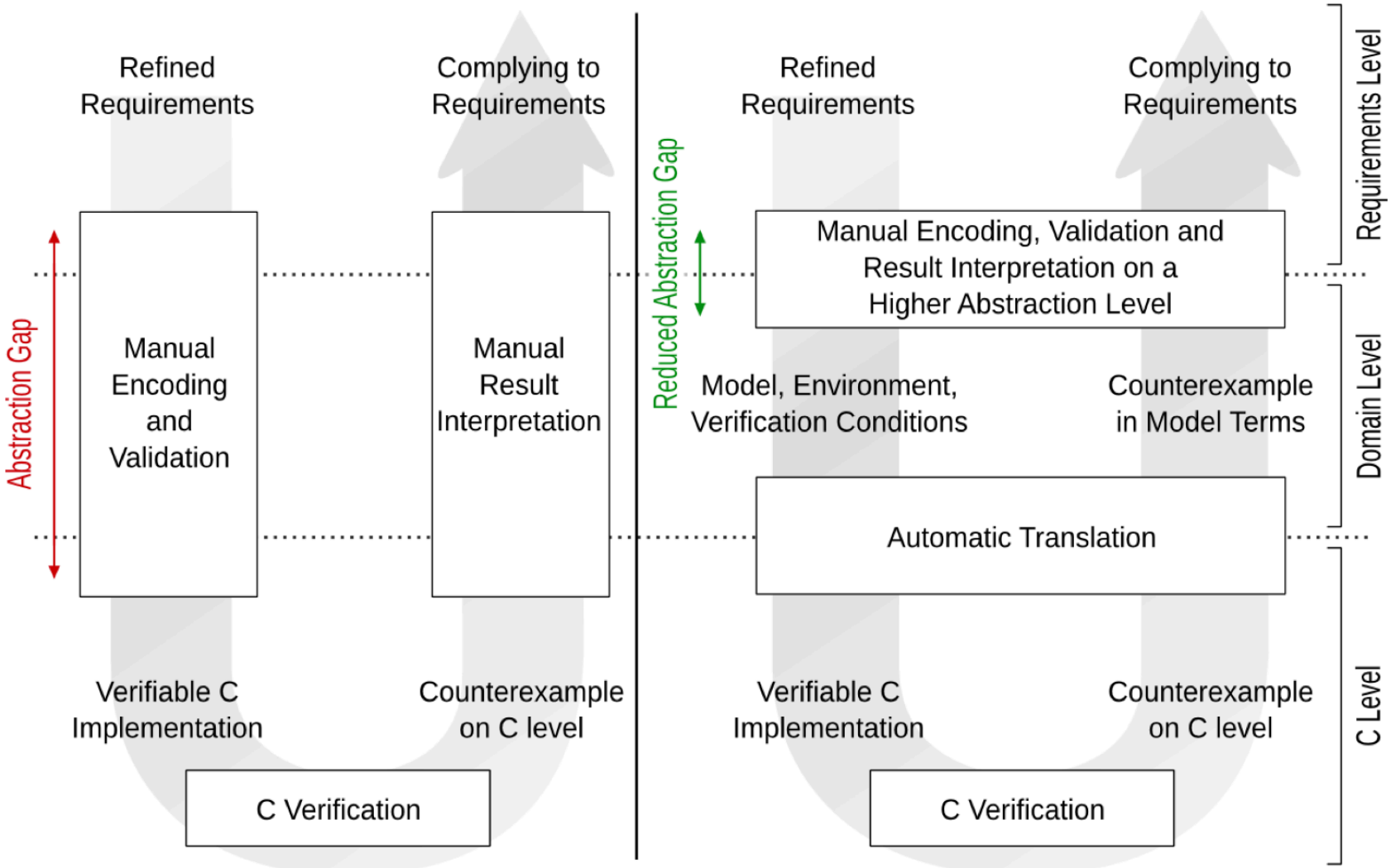
1. Make Program Expressive
2. Translate to C
3. Verify on C Level
4. Lift Results Back

DEMO



Component Contracts

[DSCV]


Refined Requirements
Complying to Requirements
Abstraction Gap
Manual Encoding and Validation
Manual Result Interpretation
Verifiable C Implementation
Counterexample on C level
C Verification
Reduced Abstraction Gap
Refined Requirements
Complying to Requirements
Manual Encoding, Validation and Result Interpretation on a Higher Abstraction Level
Model, Environment, Verification Conditions
Counterexample in Model Terms
Automatic Translation
Verifiable C Implementation
Counterexample on C level
C Verification
Requirements Level
Domain Level
C Level

[DSCV] SUV

```
statemachine VVI initial = Start
 in config(int lri, int vrp)
 in s
 in t
 out p => doPace()
 int c = 0
 int LRI = 0
 int VRP = 0

 state Start {
  on config -> Start {
   LRI = lri;
   VRP = vrp; }
  on t -> Pace
 }

state Wait {
  on s [c <= VRP] -> Wait
  on s [c > VRP] -> Wait {
   c = 0; }
  on t [c < LRI] -> Wait { ++c; }
  on t [c == LRI] -> Pace
 }

 state Pace {
  on entry {
   send p;
   c = 0; }
  on t -> Wait {++c; }
 }
```

⇨

```
typedef enum  states {
  Start,  Wait,  Pace
} VVI_States;

VVI_States state;
int c;
int LRI;
int VRP;

typedef enum events {
  config,  s,  t
} Events_VVI;
```

```
void execute(Events_VVI e,
                    int** args)
{ switch (state) {
   case Start: {
    switch (e) {
      case config: {
         LRI = *args[0];
         VRP = *args[1];
         state = Start;
         return; }
     case t: {
         state = Pace;
         doPace();
         c = 0;
         return; }
```

```
case Wait: {
 switch (e) {
  case t: {
   if ( c < LRI ) { ++c;
    return ; }
   if ( c == LRI ) {
    state = Pace;
    doPace();
    c = 0;
    return; }
  }
  case s: {
   if ( c <= VRP ) {
    state = Wait;
   }      // ...
```

[DSCV] VCs

```
after smIsInState(Wait)
  until smIsInState(Pace)
  exists c == LRI
```

⇨

```
static int q = 0;
static int e = 1;
if (state == Pace)  {
  q = 0;
  assert(e);
  e = 1;
}
```

```
if ( q )  {
 e = (c == LRI) || e;
}
if (state == Wait
        &&  !q)
{ q = 1; e = 0;  }
```

A

```
nondet_choice {
  choice {
    smtrigger(s);
  }
  choice {
    // nothing
  }
}
```

⇨

```
int ch = nondet_int();
assume(0 <= ch
            && ch < 2);
if (ch == 0) {
  execute(s, 0);
}
if (ch == 1) {
  // nothing
}
```

B

```
smStateSubset Initial:
   smInState(Pace)
     && VRP < LRI;

smNonDetInit(Initial);
```

⇨

```
state = (VVI_States)
            nondet_int();
assume(0<= state
        && state <= 2);
c = nondet_int();
LRI = nondet_int();
VRP = nondet_int();
assume(state == Pace
            && VRP < LRI);
```

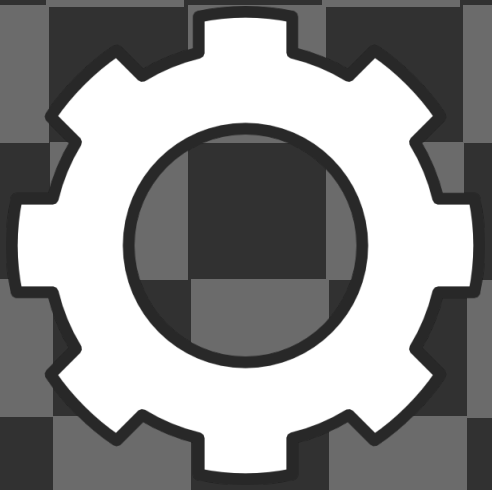
C

[DSCV] VS

```
induction on t
   from Initial
   until MAX_LRI {
     nondetermenistically smtrigger(s);
     after       smIsInState(Wait)
       until smIsInState(Pace)
         exists c == LRI }
```

```
smNonDetInit(Initial);
for (int step = 0; step < MAX_LRI; ++step) {
   nondetermenistically smtrigger(s);
   after smIsInState(Wait)
     until smIsInState(Pace)
       exists c == LRI;
   smtrigger(t);
   after step > 1 until step == MAX_LRI
      exists smIsInStateSubset(Initial); }
```

DEMO



Pacemaker

[DSCV?]

Trustworthiness of Results

Hiding Low-Level Details

Efforts for Building Extensions

End-User Workflow

Extensibility

Other Languages

[DSCV vs. X]

Code-Level Verification Tools

High-Level Spec Languages

Modeling/Verification Tools

[Validation]

(Partially formal) Requirements

4 **Points you get for each trackpoint**
InFlightPoints /functional: tags

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent feugiat enim arcu, ut egestas velit. Suspendisse potenti. Etiam risus ante, bibendum ut mattis eget, convallis sit amet nunc. Ut nec justo sapien, vel condimentum velit. Quisque venenatis faucibus tellus consequat rhoncus. Vestibulum dapibus dictum vulputate. Phasellus rhoncus quam eu dui dictum sollicitudin. Duis tempus justo magna. Nunc lobortis libero sed eros interdum aliquet ele. It uses **@req**(PointsFactor) sdf **@cfmod**(ArchitecturalComponents) to calculate the total points.

calculation PointForATrackpoint: This rule computes the points awarded for a Trackpoint. It does so by taking into account the @alt and the @speed passed as arguments.

parameters: **int16** alt: current altitude of the trackpoint
int16 speed: current speed of the trackpoint
=> (uint8 || int8)

result = (*BASEPOINTS* * 1) *

	alt > 2000	alt > 1000
speed > 180	30	15
speed > 130	10	20

otherwise 0

tests:
```
PointForATrackpoint(500, 100) == 0
PointForATrackpoint(500, 1200) == 0
PointForATrackpoint(1100, 165) == 200
PointForATrackpoint(2100, 140) == 100
PointForATrackpoint(2100, 200) == 300
```

[Validation]

Ubiquitous Tracing

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

[checked]
exported statemachine FlightAnalyzer initial = beforeFlight {
  in event next(Trackpoint* tp) <no binding>
  [in event reset() <no binding>]-> implements ExampleWithDependencies
  [out event crashNotification() => raiseAlarm]-> implements ExampleWithDependencies
  readable var int16 points = 0
  state beforeFlight {
    entry { points = 0; }
    on next [tp->alt > 0 m] -> airborne
    [exit { points += TAKEOFF; }]-> implements PointsForTakeoff
  } state beforeFlight
```

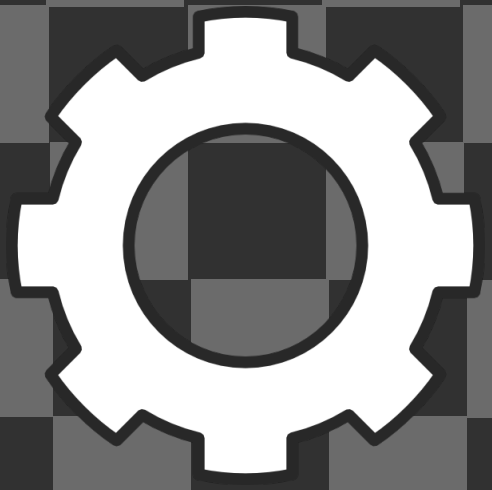
[Validation]

... also from Verification Conditions

```
//[ Assures that the atria are paced timely ]
after timesAP > 0 =>                          -> implements atrialPacing
  lastTimeA - lastTimeV <= AEI - 6;
```

Nicely integrated V&V

DEMO



Tracing

[End User]

Easier to write verifiable code.

Easier to write the VCs.

Results are more understandable.

Some/many tool details hidden.

=> Potentially increases the use of verification in practice.

[Lang Developer]

Identify interesting properties.
Make them easily expressible.
Integrate the analyses (tools)
Lift the Results.

=> A systematic approach for integrating V&V into the IDEs

[Meta Level]

How do we verify the languages, tools, transformations?

Currently: testing with enough coverage is „good enough“.

=> In the future, a more formal way of lang defintion is needed.

5

Opportunities & Challenges

Good Experience.

Neutral Observation

Problem/Challenge

Default extensions are useful, in particular components, state machines and units.

Easy and useful to add customer/project specific extensions.

The Tracing problem is solved – at least from a technical perspective.

Decided not to make extensions BL independent, they are actually C extensions and cannot be used with other base languages.

Integration with analysis tools work and is useful, but performance and config of the analysis is still an issue. (leaky abstraction)

Do more verification on code level than on model level because of consistency problem with code.

Verification should be done incrementally – Verification-driven Development (VDD)

Splitting C into several languages not so useful – dependencies!

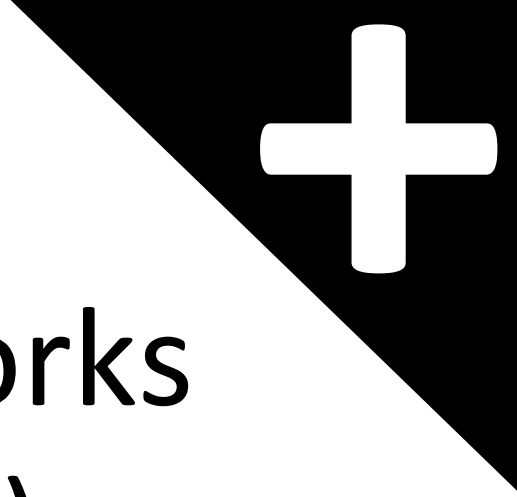
Modularity works in principle and practice

MPS‘ approach scales to non-trivial and many languages.

Flexible notations actually work and are useful in practice.

Decoupling Notation from Language works.

MPS is easily extensible with new notational styles.



VCS integration works well (diff/merge)

MPS can be extended with the same means – bootstrapped.

MPS also supports debugging of DSLs – even though we had to extend the mechanism

No direct support for detecting semantic interations between languages

Modularity:
Sometimes base language requires change (introduction of abstract class or interface)

Ability to create additional language aspects missing (you can existing ones)

Debugger definition separate from generator; leads to duplication

many aspects of language definition too „procedural" and hence hard to analyze.

Due to the open world assumption of MPS, there is a "feeling of incompleteness“ in aspects like e.g. in lifting analyses results.

Thank you!

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