Testing Languages, Generators and Runtimes

for a Safety-Critical System

Markus Völter

with Bernd Kolb, Klaus Birken, Federico Tomassetti, Patrick Alff, Laurent Wiart

voelter ingenieurbüro für softwaretechnologie // itemis

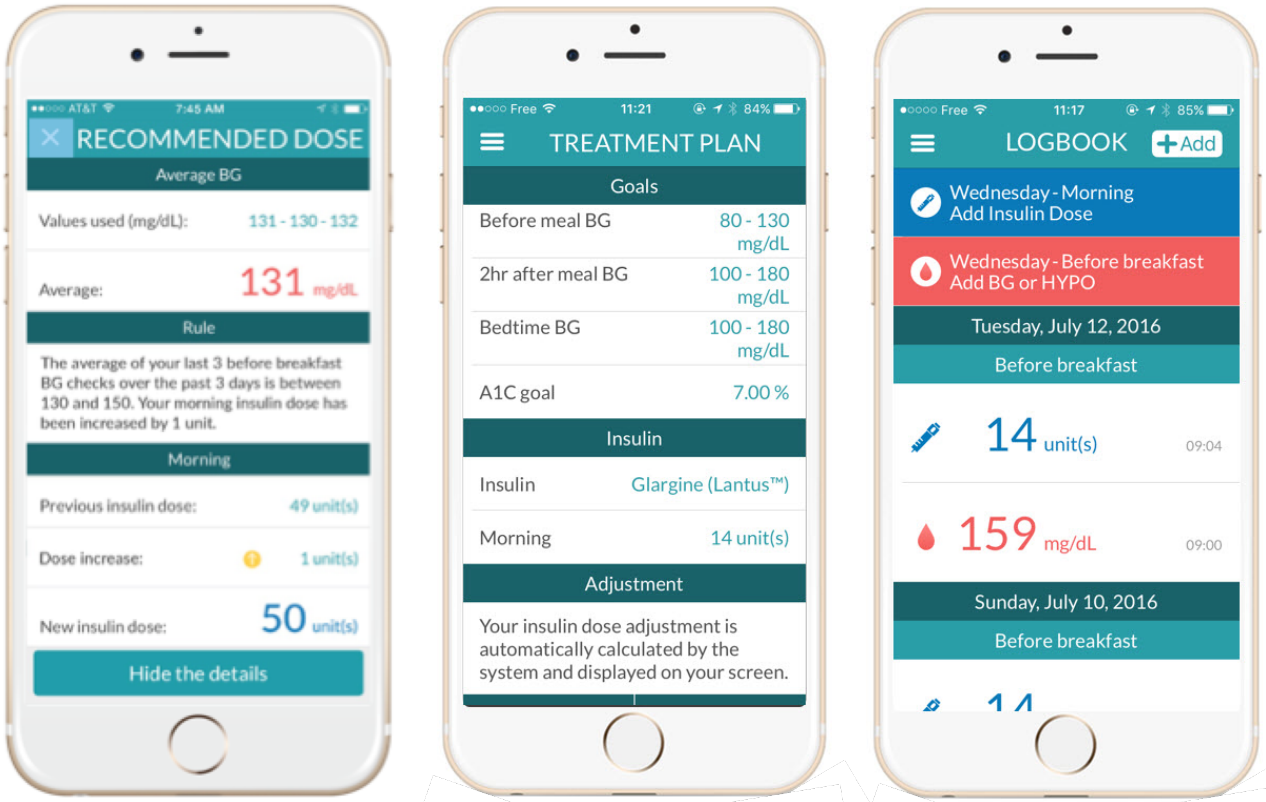
voelter@acm.org
www.voelter.de
@markusvoelter

Context

Mobile Apps that help patients w/ treatments

Monitor side-effects and recommend actions

Manage dosage of medications



VOLUNTIS
Connected Therapeutics

Context

Mobile Apps that help patients w/ treatments

Monitor side-effects and recommend actions

Manage dosage of medications

“Algorithms“ for recommendations and dosage at the core of these apps.

Safety-critical, since they could hurt patients.

FDA

Customer develops many different apps/algos like this, efficiency of algo development is key.

VOLUNTIS
Connected Therapeutics

Solution Approach

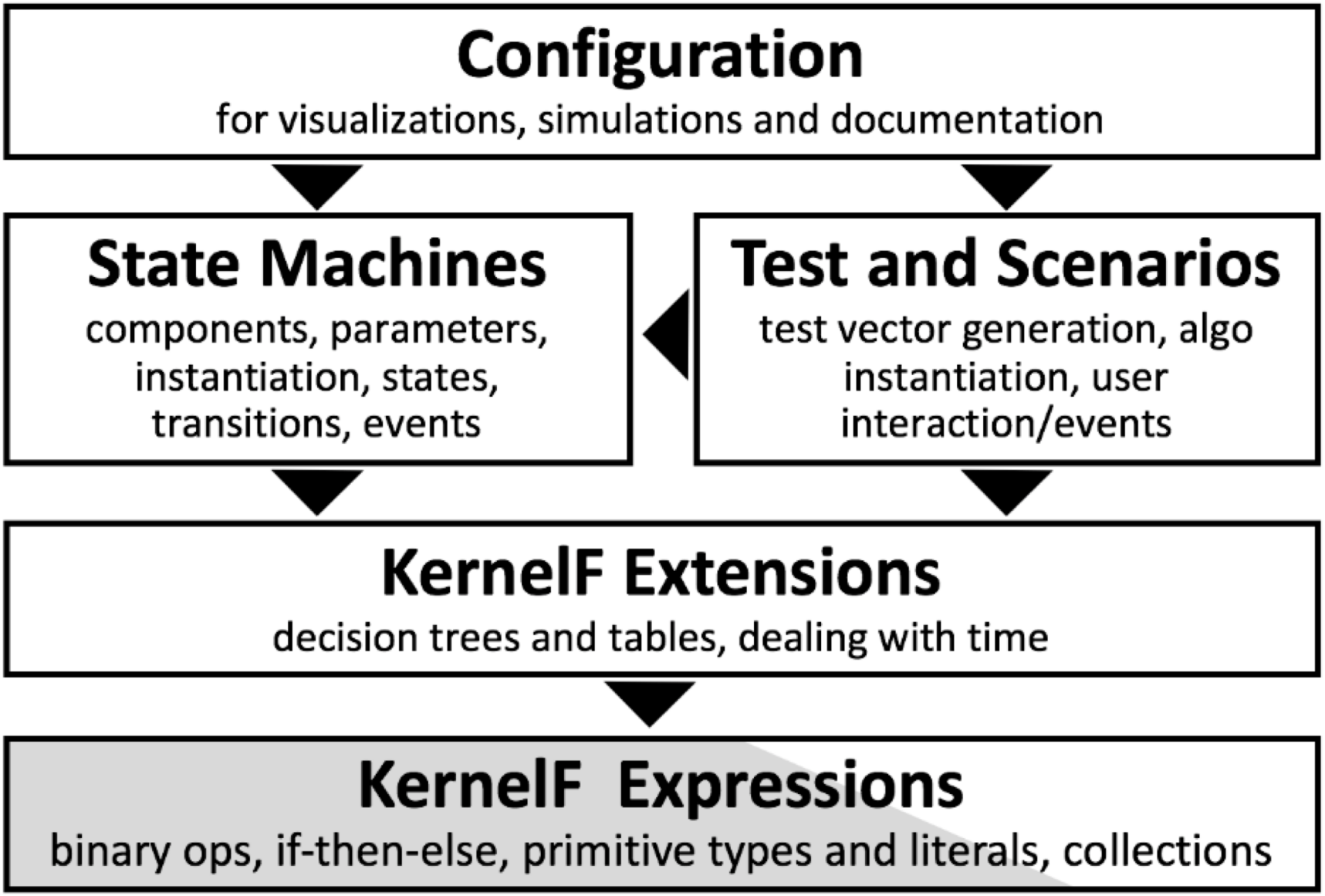
Health care professionals directly „code“ algos, using a suitable language.

Avoids indirections through requirements docs.

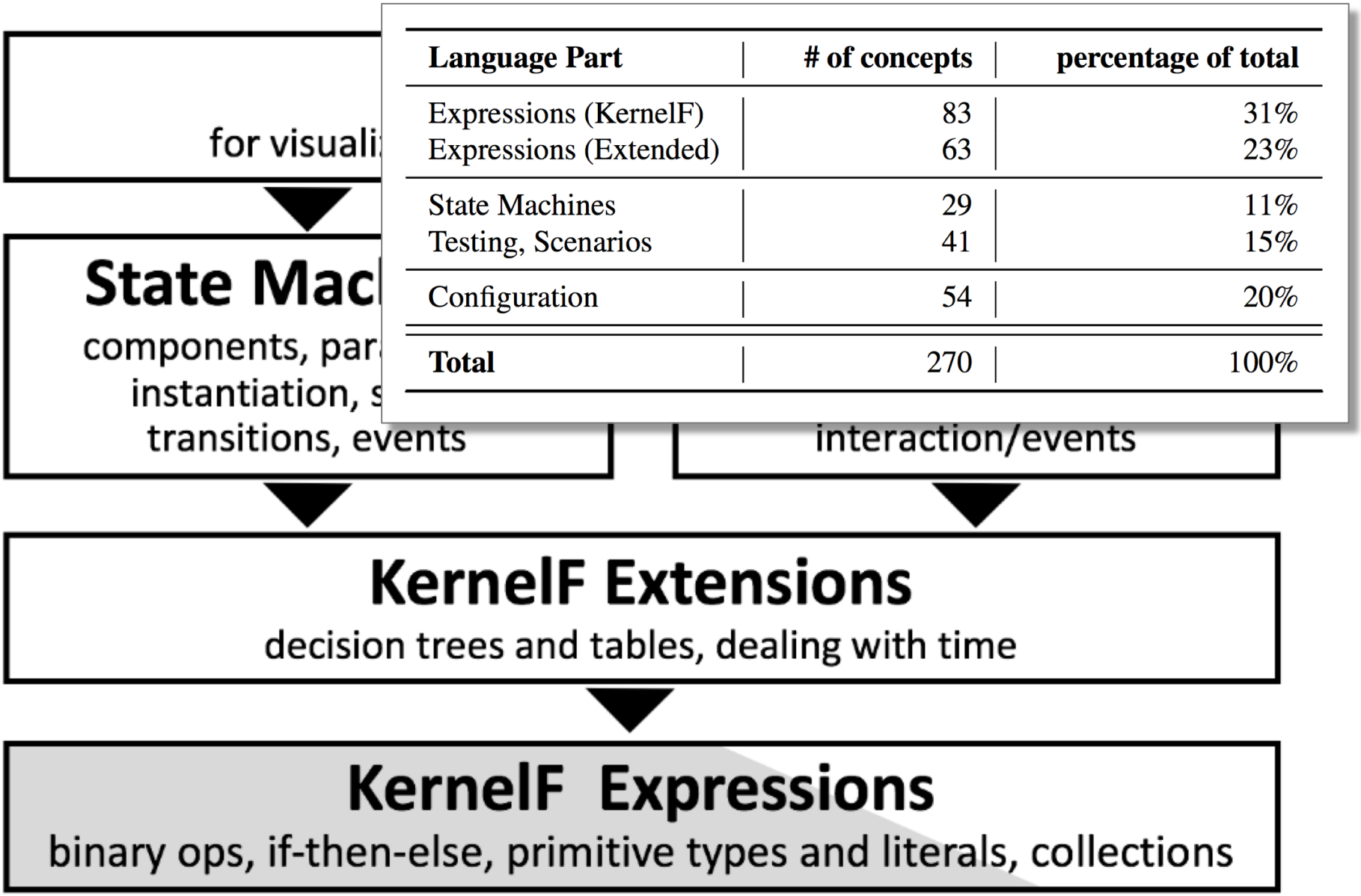
Speed up dev significantly.

} Pretty typical DSL-based dev-approach.

Languages Used



Languages Used



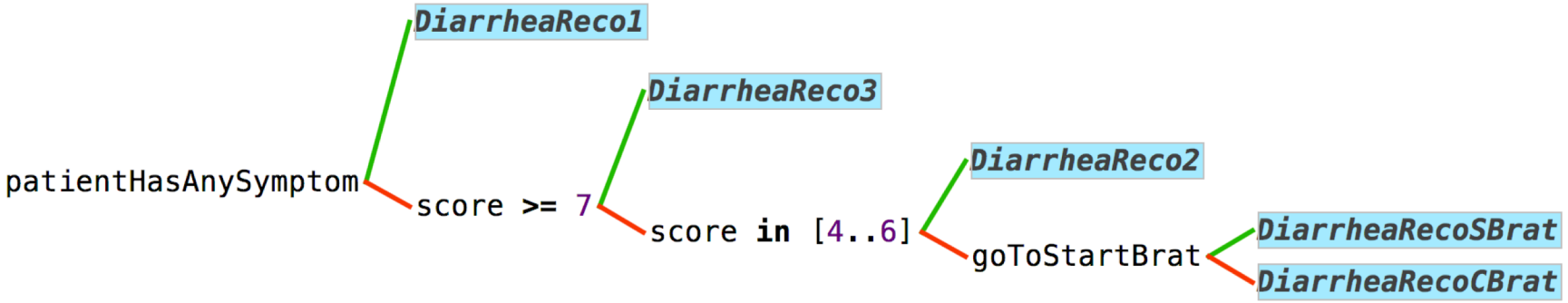
Language Part	# of concepts	percentage of total
Expressions (KernelF)	83	31%
Expressions (Extended)	63	23%
State Machines	29	11%
Testing, Scenarios	41	15%
Configuration	54	20%
Total	270	100%

Some Language Impressions I

```
decision table BpScoreDecisionTable(sys: bpRange, dia: bpRange) =
```

		dia					
		<= 50	[51..90]	[91..95]	[96..100]	[101..109]	>= 110
sys	<= 90	1	1	3	4	5	6
	[91..140]	2	2	3	4	5	6
	[141..150]	3	3	3	4	5	6
	[151..160]	4	4	4	4	5	6
	[161..179]	5	5	5	5	5	6
	>= 180	6	6	6	6	6	6

```
decision tree DiarrheaStoolsDecisionTree(score: DiarrheaStoolsOverBaseline,
                                         patientHasAnySymptom: boolean, goToStartBrat: boolean)
```




```
type temperature: number[36|42]{1}
type measuredTemp: number[35|43]{2}

Error: type number[32.55|39.99]{4} is not a subtype of number[36|42]{1}
val T_measured: measuredTemp = 42.22
val T_calibrated: temperature = T_measured * 0.93
```

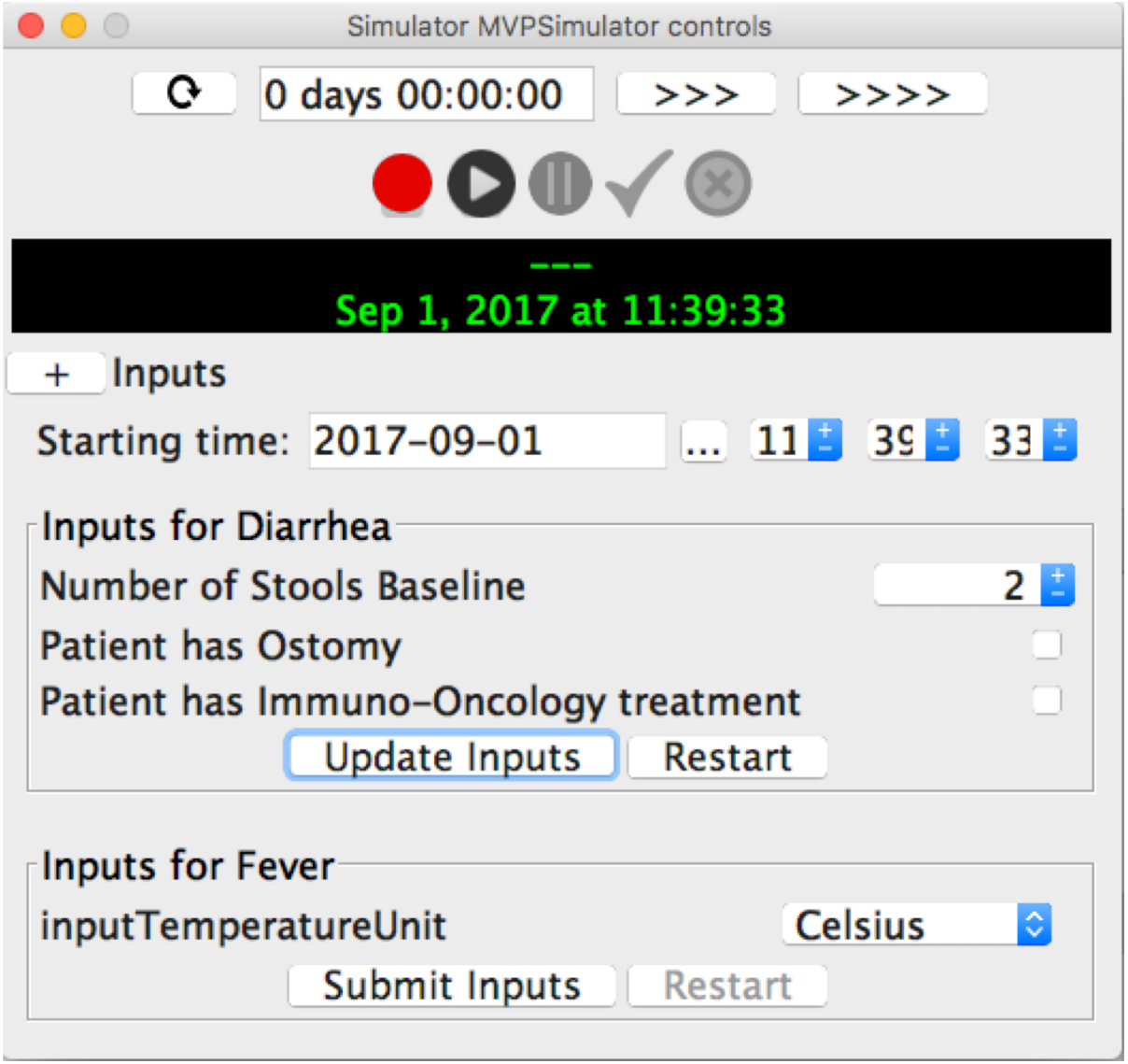
Some Language Impressions II

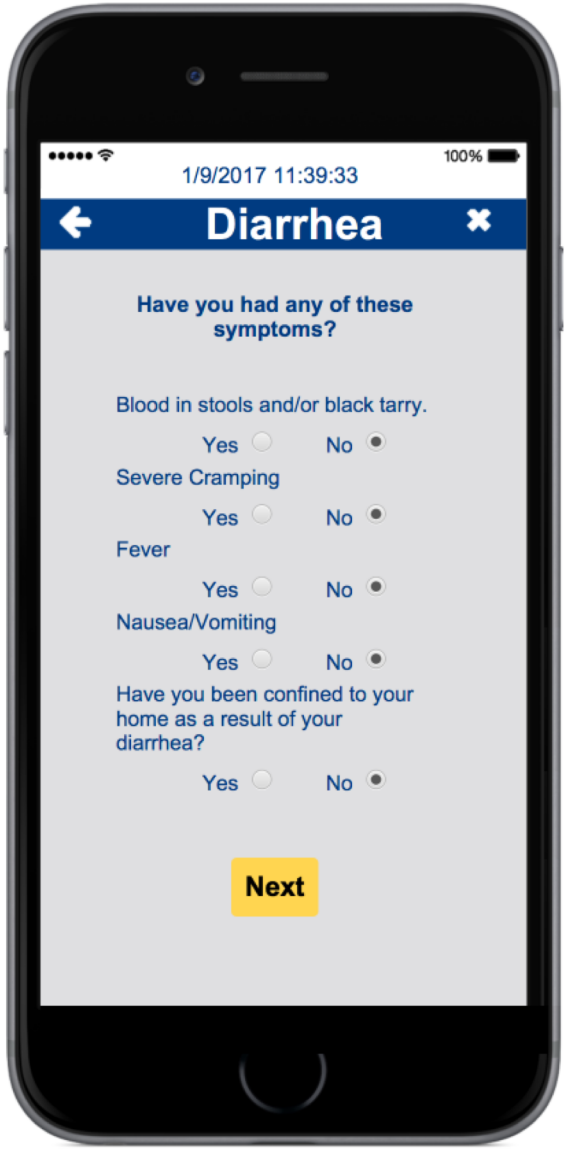
```
PASS
function test gradeStools
  given  7  expected  3
  given  6  expected  2
  given  5  expected  2
  given  4  expected  2

PASS
function test DiarrheaStoolsDecisionTree
  given  false, 1, true, false    expected   DiarrheaUSRecoLevel1Symptom
  given  false, 9, false, false   expected   DiarrheaUSRecoGrade3

PASS
function test checkScreeningQuestion
  given   answers to DiarrheaScreeningQuestionnaire{    expected  true
            dietarySupplements: false
            medication        : true
            hospitalized      : false
          }
```

Some Language Impressions III





Solution Approach

Health care professionals directly „code“ algos, using a suitable language.

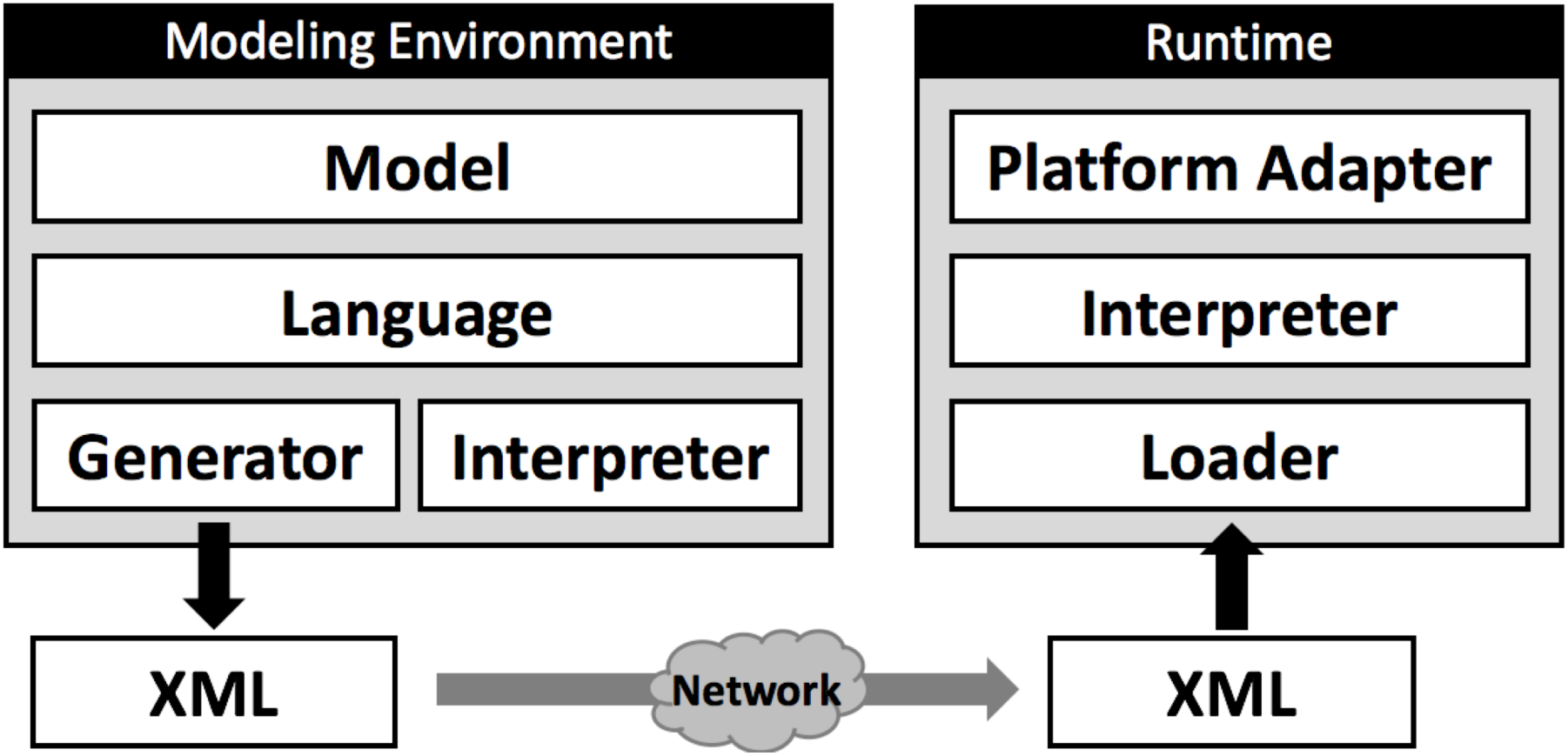
Avoids indirections through requirements docs.

Speed up dev significantly.

} Pretty typical DSL-based dev-approach.

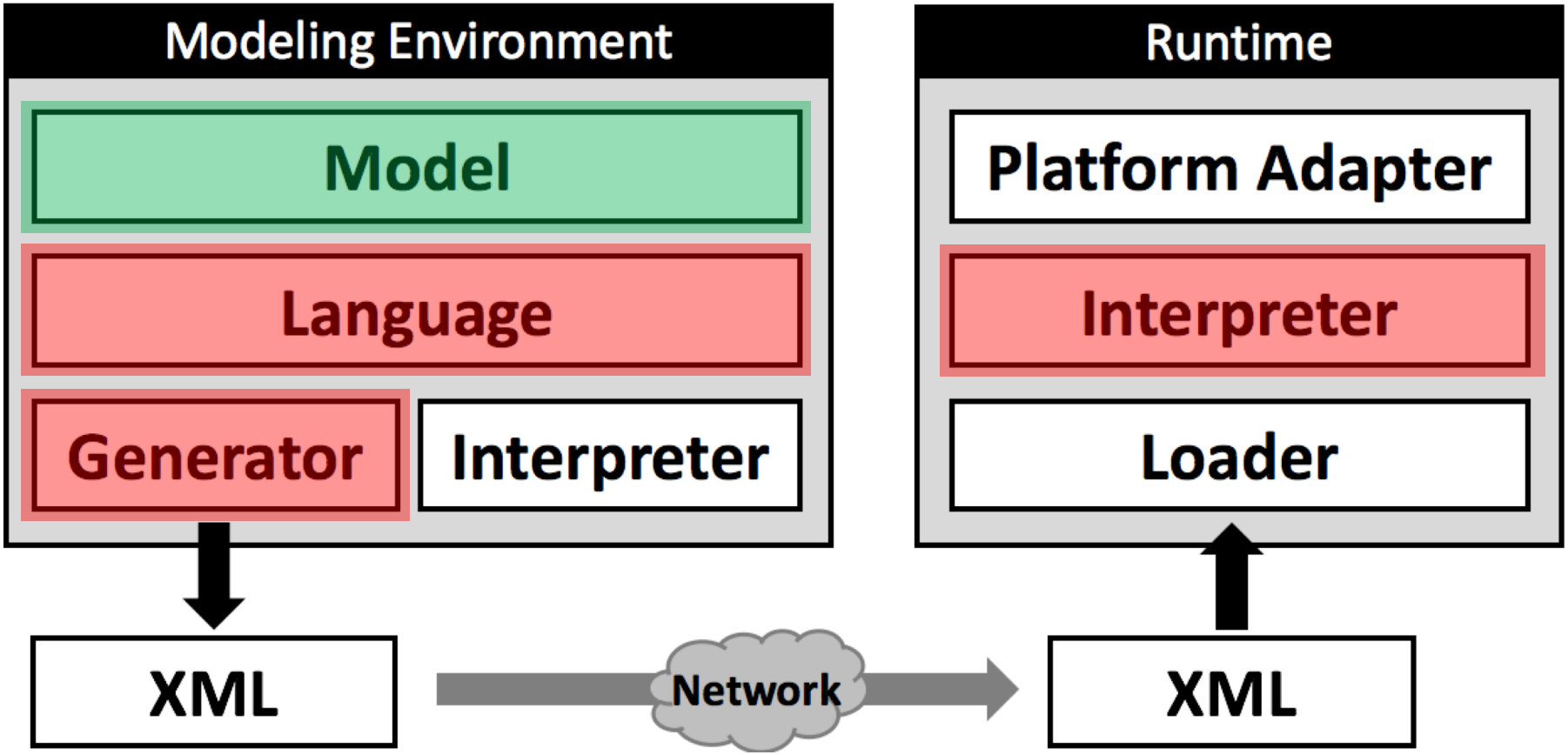
What good is all the abstraction if we cannot trust the translation to the implementation?

System Architecture



What good is all the abstraction if we cannot trust the translation to the implementation?

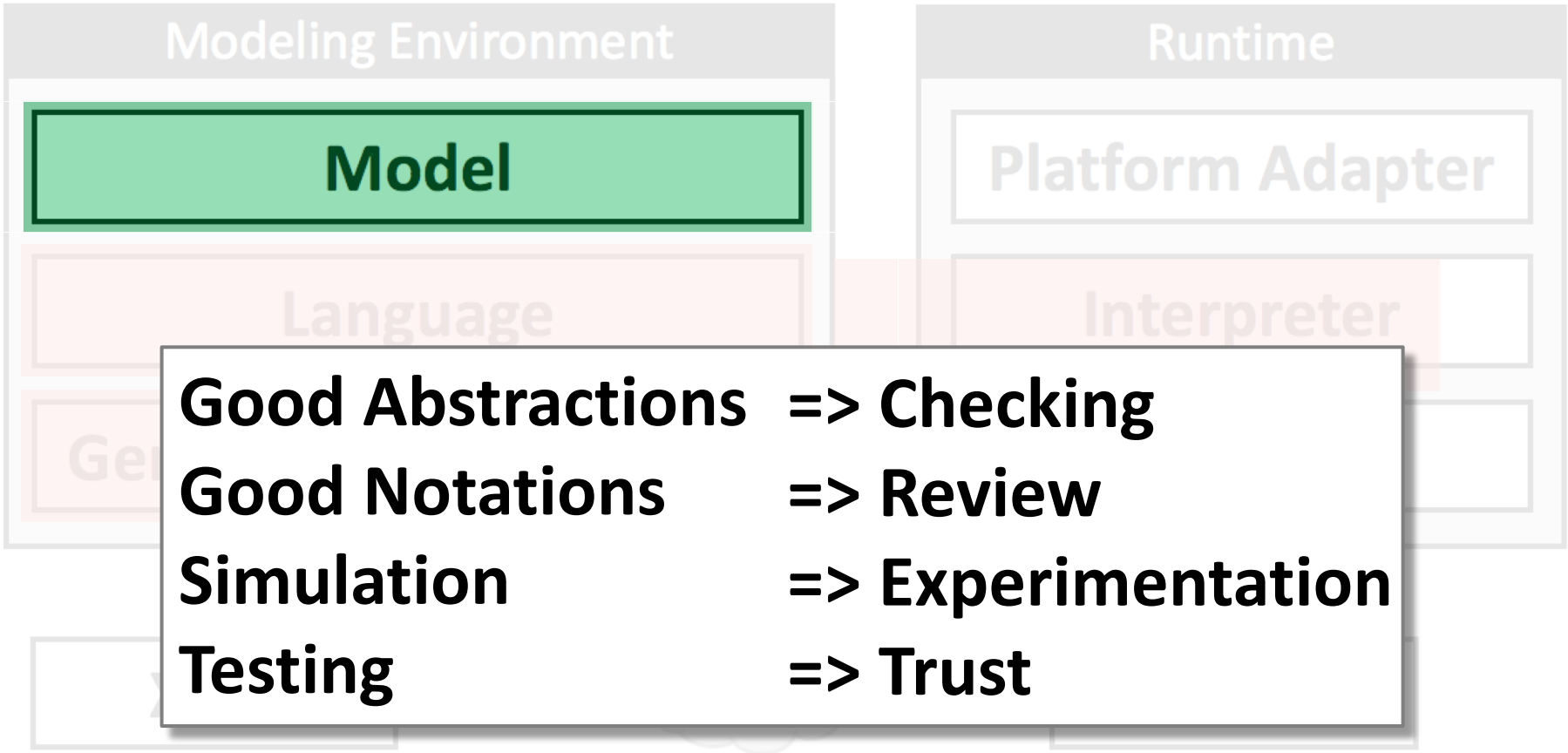
System Architecture & Safety Standards



Tools may introduce *additional systematic errors* if faulty.
Safety **standards** require reliable mitigation of such errors.

DO-178C EN50129 IEC62304 ISO26262

System Architecture & Safety Standards



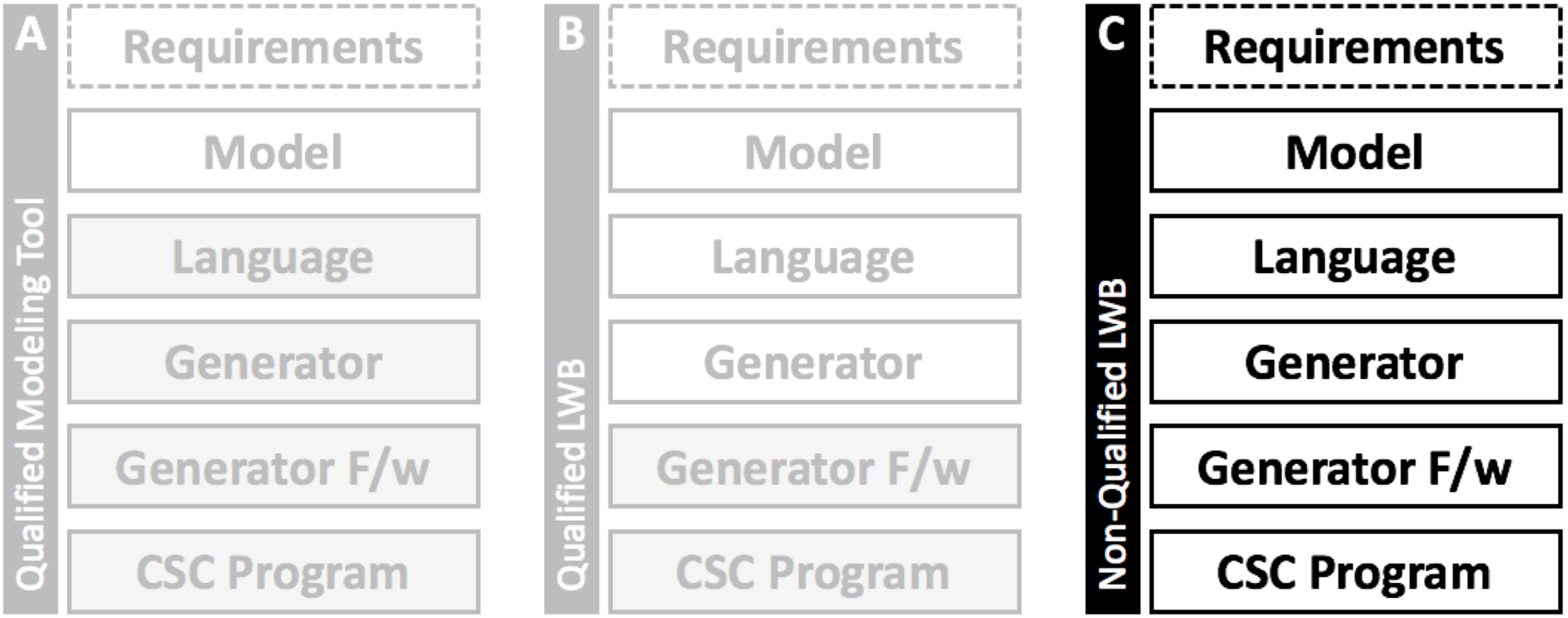
Good Abstractions => Checking
Good Notations => Review
Simulation => Experimentation
Testing => Trust

Tools may introduce *additional systematic errors* if faulty.
Safety **standards** require reliable mitigation of such errors.

DO-178C EN50129 IEC62304 ISO26262

Unqualified Tools!

What good is all the abstraction if we cannot trust the translation to the implementation?

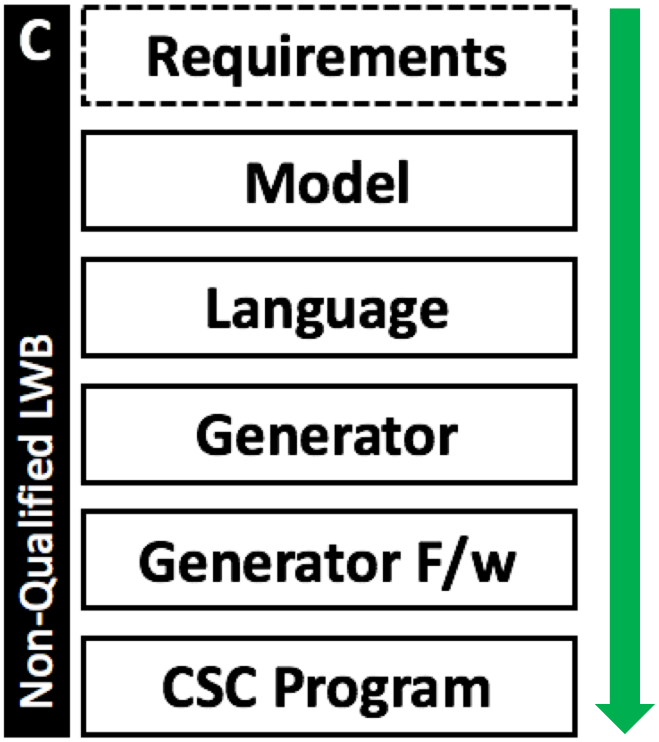


Unqualified Tools!

End-to-end testing required.
How to do this without exploding effort?

Automated Redundancy

catch errors in redundant path
while reducing manual effort.



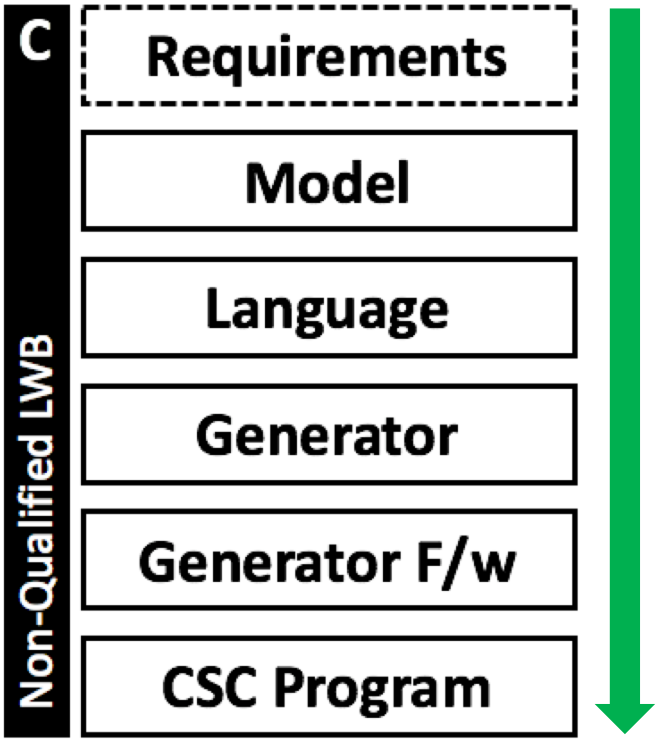
Unqualified Tools!

End-to-end testing required.
How to do this without exploding effort?

Automated Redundancy

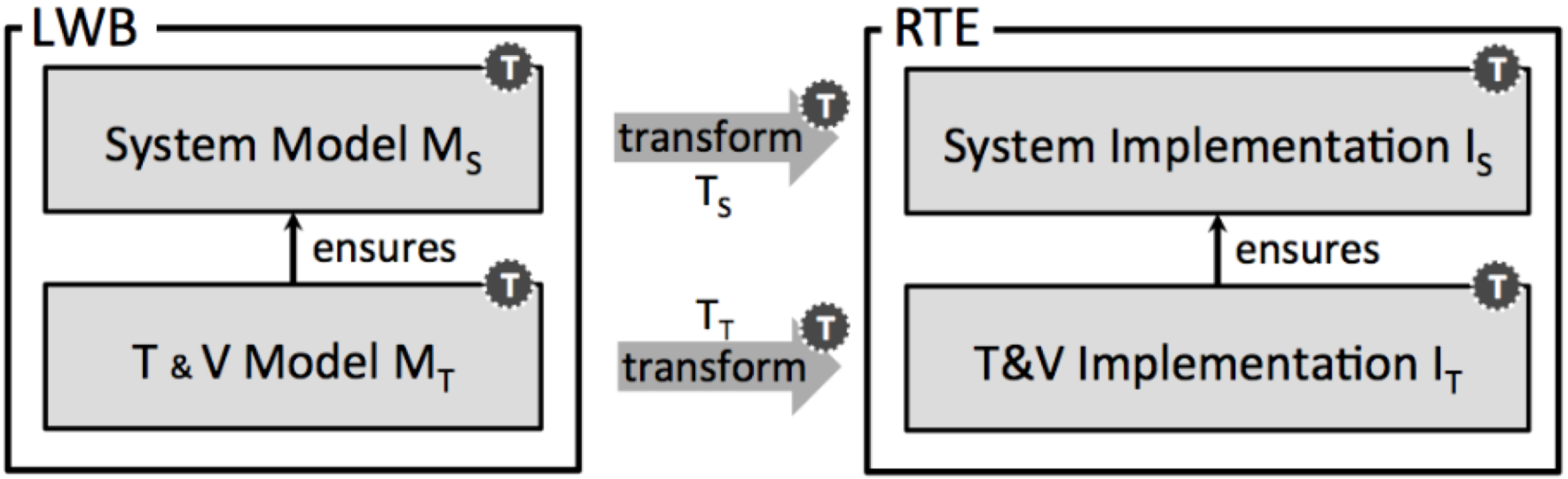
catch errors in redundant path
while reducing manual effort.

+ specific risk mitigations



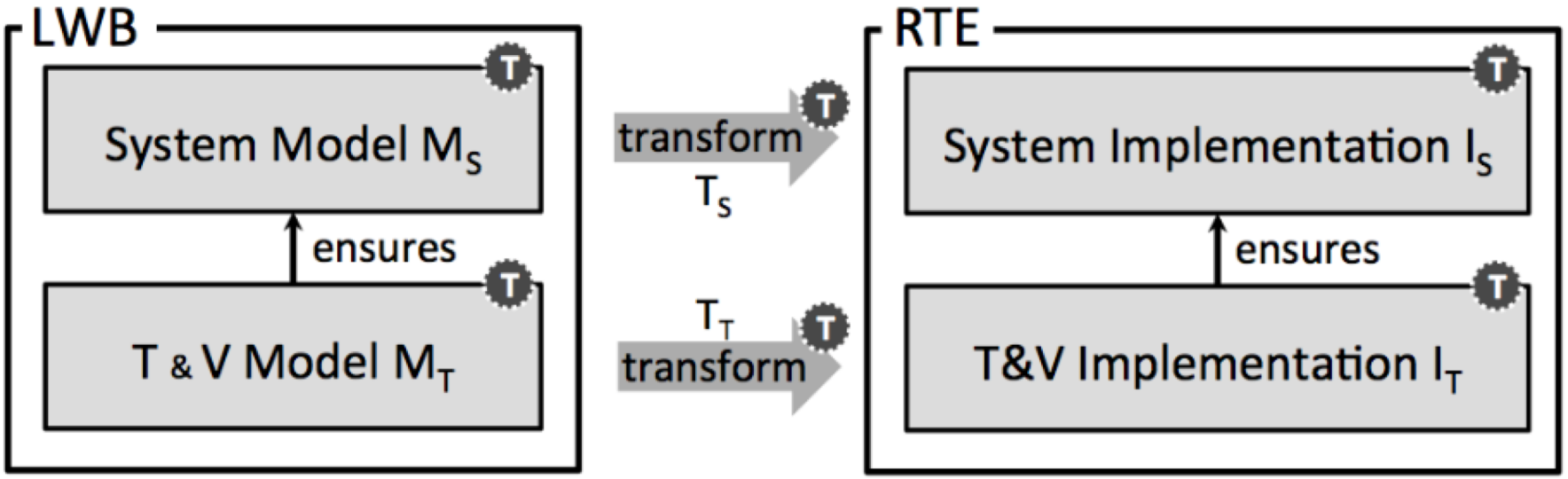
Modeling Architecture

Model the Algo/System with the DSL and also model the tests/verification. Then translate both and execute on the level of the implementation.



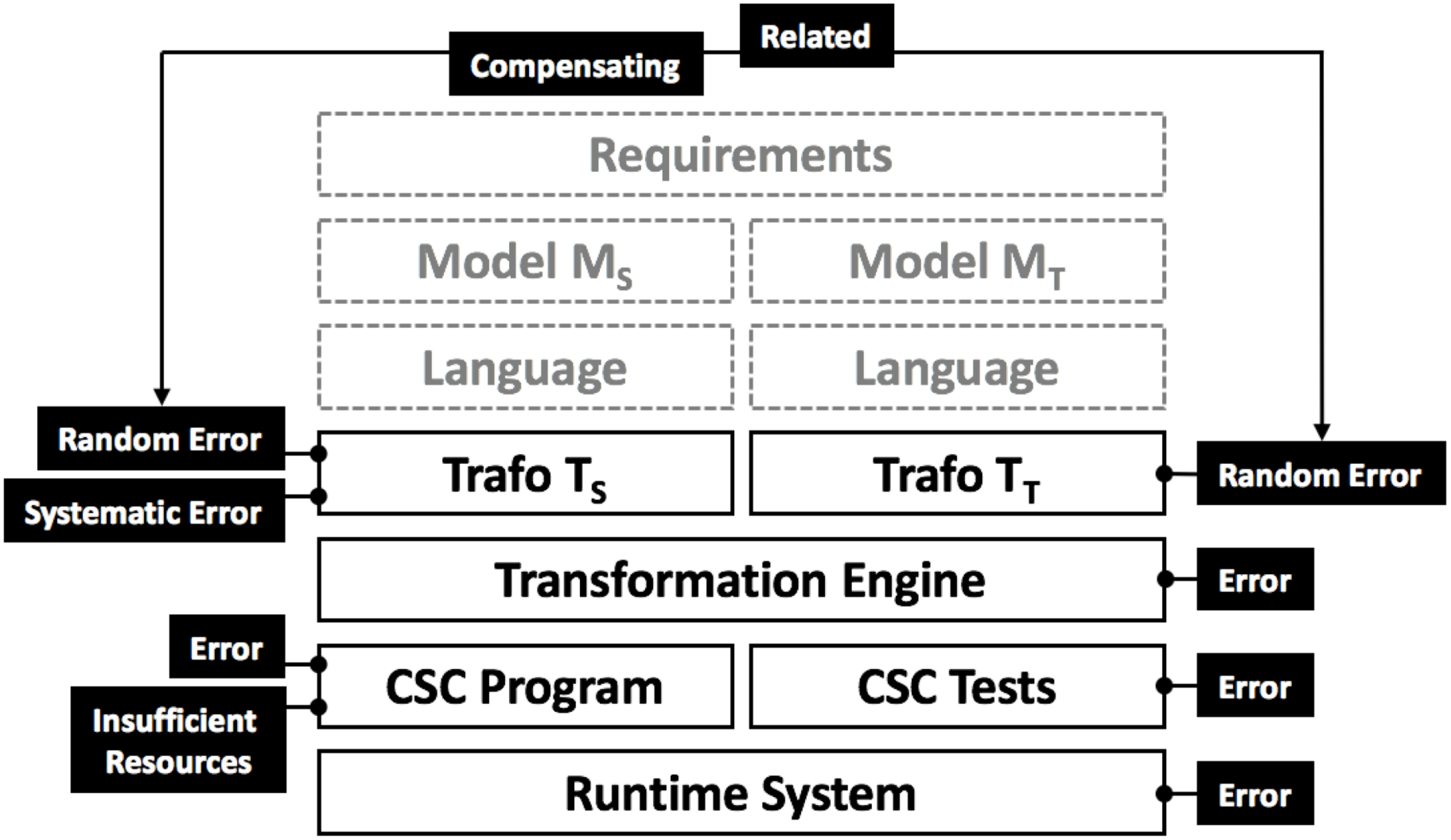
Modeling Architecture

Model the Algo/System with the DSL and also model the tests/verification. Then translate both and execute on the level of the implementation.

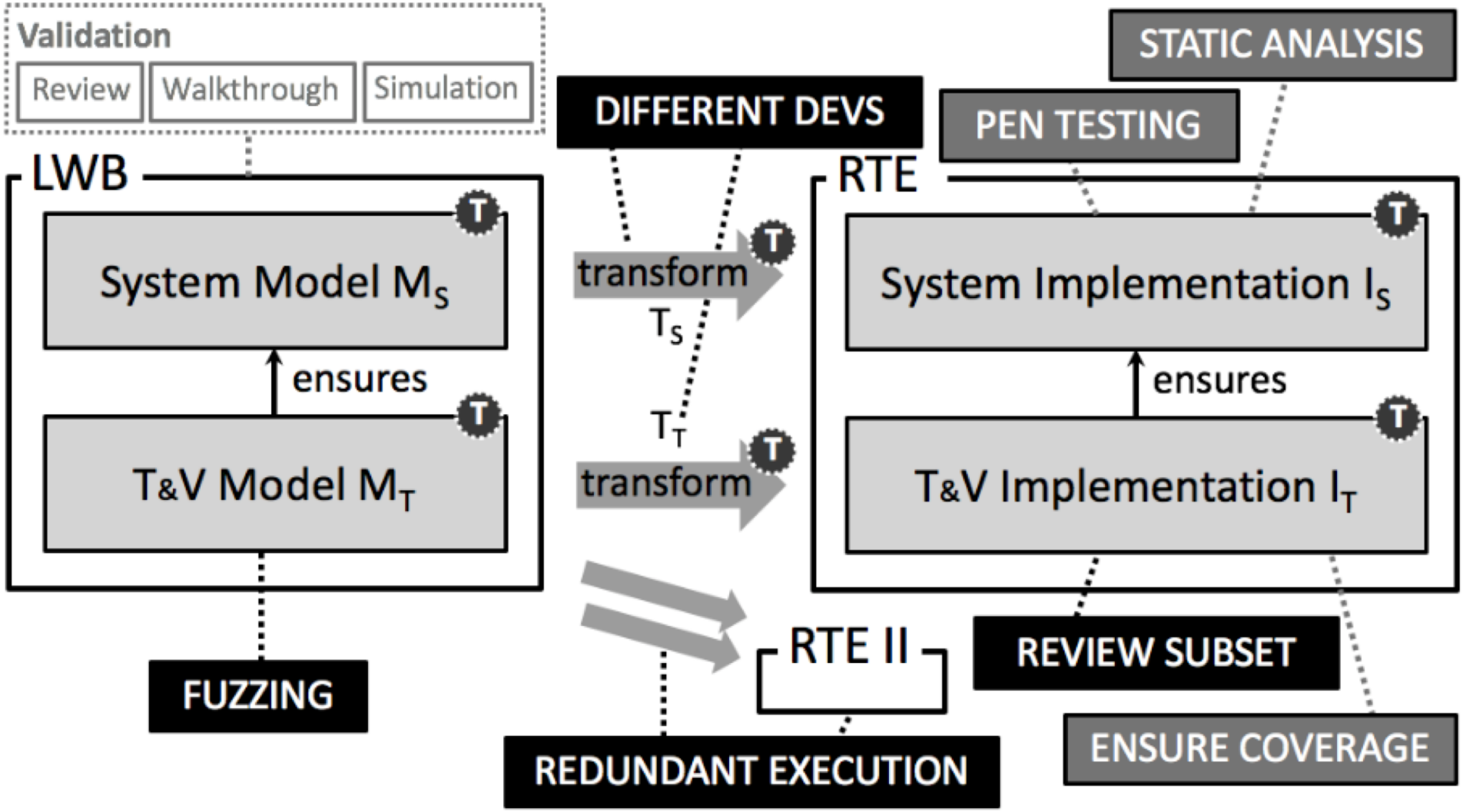


+ Risk Analysis + Mitigations

Risk Analysis



Mitigations – Safe Modeling Architecture



Mitigations – Safe Modeling Architecture

use redundant execution on two execution engines
use different developers for the two trafos
review a subset of the generated code
clearly define and QA the DSL

---

only these specific to DSL use

to use fuzzing on the tests
ensure high coverage for the tests
run the tests on the final device
perform static analysis on the generated code
perform penetration testing on the final system
and use architectural safety mechanisms.

Mitigations – Safe Modeling Architecture

use **redundant** execution on two execution engines

- **C++ interpreter on device**
- **In-IDE Java Interpreter**

Lots of overhead? Not really.

Validation: the in-IDE interpreter is used for interactive testing, exploration, understanding, simulation. HCP‘s single-most appreciated use of the models!

Verification: addresses unrelated but compensating, as well as related errors in the transformations. Does not rely on trafo engine, so finds error in it. It‘s also simple (!fast), so acts as a specification.

Test Stats and other Numbers

Two reference Algos, 305 test cases for Bluejay, 297 for Greenjay, plus lower-level tests for decision tables and trees

100% line coverage regarding language structure, Java interpreter and C++ interpreter

Validation Effort Reduction from **50 PD to 15 PD**

Test Setup Effort reduced by a **factor of 20**

Shortened Turnaround for req -> impl -> write tests -> execute tests b/c of much better tool integration

„Tremendous Speedup“ for changes to algo *after* it has been validated – automatic reexecution of everything.

Coverage Analysis

kernelf
functional • embeddable • extensible

```
assessment: StructuralCoverage
query:      structural test coverage {...}
sorted: ☒    must be ok: ☐    hide ok ones: ☐
last updated: Feb 28, 2017 (9 months ago) by markusvoelter
```

---

org.iets3.core.expr.base

TupleValue	Covered. N=2, V=12 H=3..3
SomeValExpr	Covered. N=11, V=197 H=6..9
LogicalImpliesExpression	Covered. N=4, V=10 H=2..3
ErrorLiteral	Covered. N=27, V=198 H=0..9
ErrorExpression	Covered. N=9, V=24 H=1..3
RangeTarget	Covered. N=13, V=104 H=4..7
TupleAccessExpr	Covered. N=4, V=10 H=2..3
SomeExpression	Covered. N=21, V=252 H=4..9
NoneExpression	Covered. N=21, V=58 H=1..6
CastExpression	Covered. N=3, V=12 H=3..6
AsSetOp	Covered. N=5, V=23 H=3..8
SimpleSortOp	Covered. N=5, V=22 H=3..5
ForeachOp	Missing.
OldMemberRef	Covered. N=1, V=32 H=8..8
RecordMember	Partial. Missing: [type_old, type_old]
RecordLiteral	Partial. Missing: [type_old]

Test Generation

kernelf
functional • embeddable • extensible

For everything that can be seen as taking a list of arguments, those can be synthesized.

```
test case TestFunctions [success] {
  vectors function add  -> producer: random 25
          results: true
```

	a	a2	b	c	s	res	status
0: valid	3	-3.06	false	BLUE	""	3	ok
1: valid	2	2.74	false	BLUE	"M/Yh-0I/ac"	2	ok
2: valid	1	0.22	false	BLUE	""	1.22	ok
3: invalid input	4	-0.45	true	BLUE	"7l:6h7sg!afLmULGU8wtI00H9"		not executed
4: invalid input	1	1.38	false	RED	"Mtoa7J66uuWTye2f2-fLhD§hj8C2K"		not executed
5: invalid input	1	-2.63	false	BLUE	"n66r7E (f0J$aQMj§"		not executed

```
vectors function plus -> producer: eqclass
        results: true
```

	a	b	c	res	status
0: valid	0	-10	GREEN	-10	ok
1: valid	0	-10	BLUE	-10	ok
2: valid	0	10	GREEN	10	ok
3: valid	0	10	BLUE	10	ok
4: valid	0	-1	GREEN	-1	ok

	a	b	status
0: valid	7	2	no expected value given; actual was 14
1: valid	1	8	no expected value given; actual was 8
2: valid	5	2	[POST] res == a * b
3: valid	7	8	no expected value given; actual was 56
4: valid	5	6	[POST] res == a * b
5: valid	8	0	[PRE] b > 0
6: valid	5	8	[POST] res == a * b

Mutation / Fuzzing

kernelf
functional • embeddable • extensible

For a set of tests that all succeed, if after a change to the program they still do, this is a problem.

```
fun doodle(a: number[1|5]) = if true then a else a * 1

fun add(a: number[1|5]) = alt [ a > 3      => a + 1
                              | otherwise => doodle(a) ]

test case TestFunctions [incomplete] {
  vectors function add                          -> producer: random 30
          results: true                            { 30 entries }
          mutator: # of mutations 20
                   keep all:      false
                     -> ParensExpression
                     -> NumberLiteral
                     -> LogicalNotExpression
                     -> ParensExpression
                     -> ParensExpression
}
```

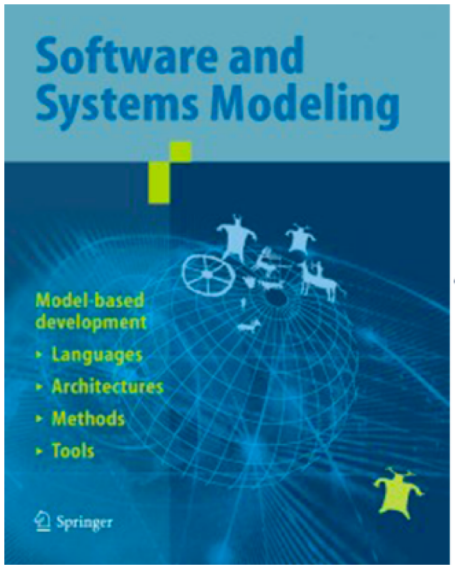
```
fun add(a: number[1|5]) = alt [ ((a) + 1) > 3 => a + 1
                              | a
                              | otherwise     => doodle(a) ]

fun doodle(a: number[1|5]) = if true then a else a * 2
                                                    | 1

fun doodle(a: number[1|5]) = if true then a else ((a * 1) + 1)
                                                  | a * 1

fun doodle(a: number[1|5]) = if !(true) then a else a * 1
                             | true
```

WRAP UP



Software & Systems Modeling
pp 1–24 | Cite as

Using language workbenches and domain-specific languages for safety-critical software development

Authors | Authors and affiliations

Markus Voelter, Bernd Kolb, Klaus Birken, Federico Tomassetti, Patrick Alff, Laurent Wiart, Andreas Wortmann, Arne Nordmann

Regular Paper
First Online: 17 May 2018

Abstract

Language workbenches support the efficient creation, integration, and use of domain-specific languages. Typically, they execute models by code generation to programming language code. This can lead to increased productivity and higher quality. However, in safety-/mission-critical environments, generated code may not be considered trustworthy, because of the lack of trust in the generation mechanisms. This makes it harder to justify the use of language workbenches in such an environment. In this paper, we demonstrate an approach to use such tools in critical environments. We argue that models created with domain-specific languages are easier to validate and that the additional risk resulting from the transformation to code can be mitigated by a suitably designed transformation and verification architecture. We validate the approach with an industrial case study from the healthcare domain. We also discuss the degree to which the approach is appropriate for critical software in space, automotive, and robotics systems.

http://voelter.de/data/pub/MPS-in-Safety-1.0.pdf

Successfully passed FDA Pre-Submission

DSLs enable non-programmers
to contribute directly to software systems.

The risks introduced by the trafos
can be clearly identified and mitigated.

Automated Redundancy
provides safety with limited additional effort.

The additional efforts can be amortized
over a few years, while speeding up dev and incr quality

A non-qualified tool is not a showstopper
End-to-end testing is feasible.

Redundant in-IDE interpreter is useful
for validation and verification.