Automated VDM-SL to Smalltalk Code Generators for Exploratory Modeling

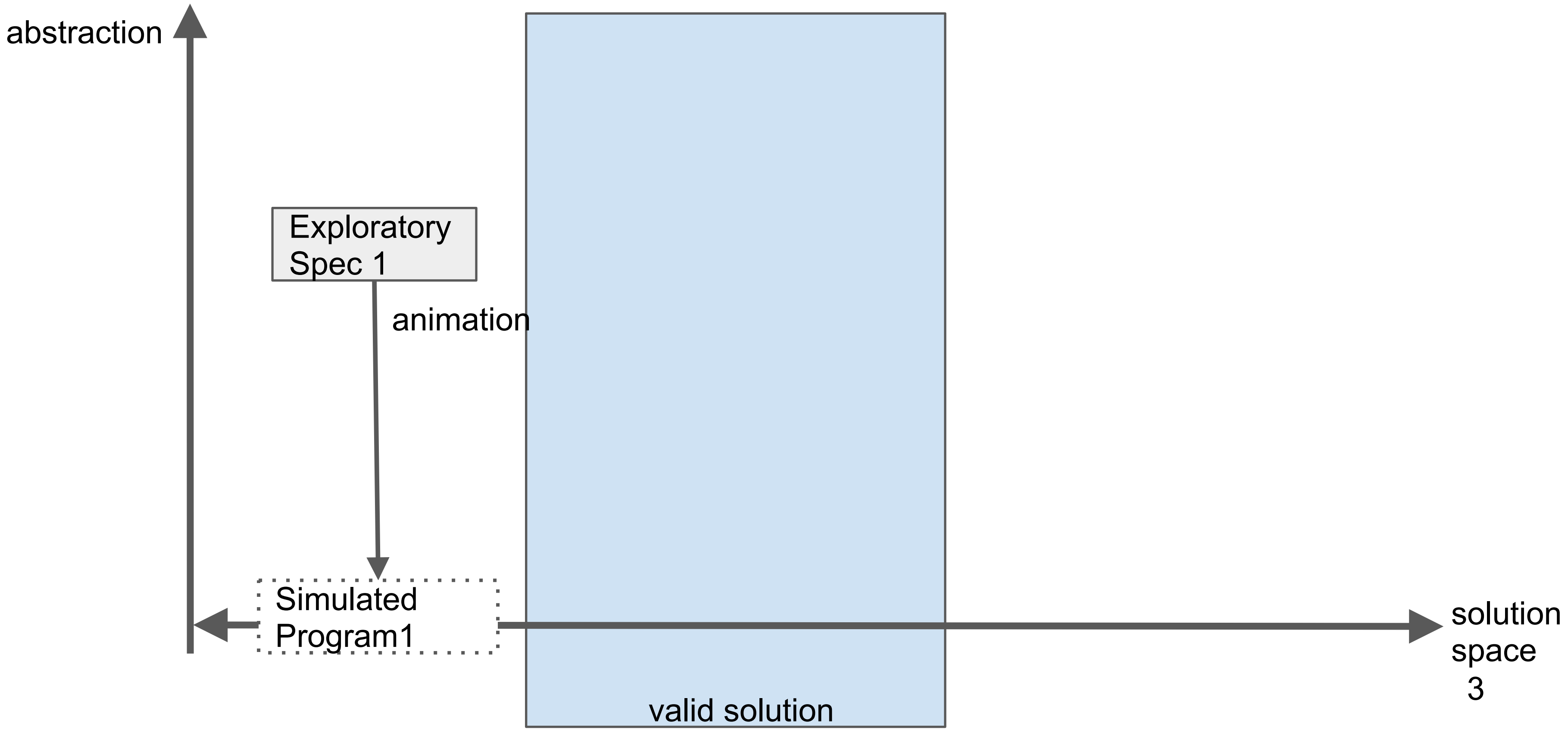
Tomohiro Oda[1,2]
Keijiro Araki[2]
Peter G. Larsen[3]

[1] Software Research Associates, Inc.
[2] Kyushu University
[3] Aarhus University

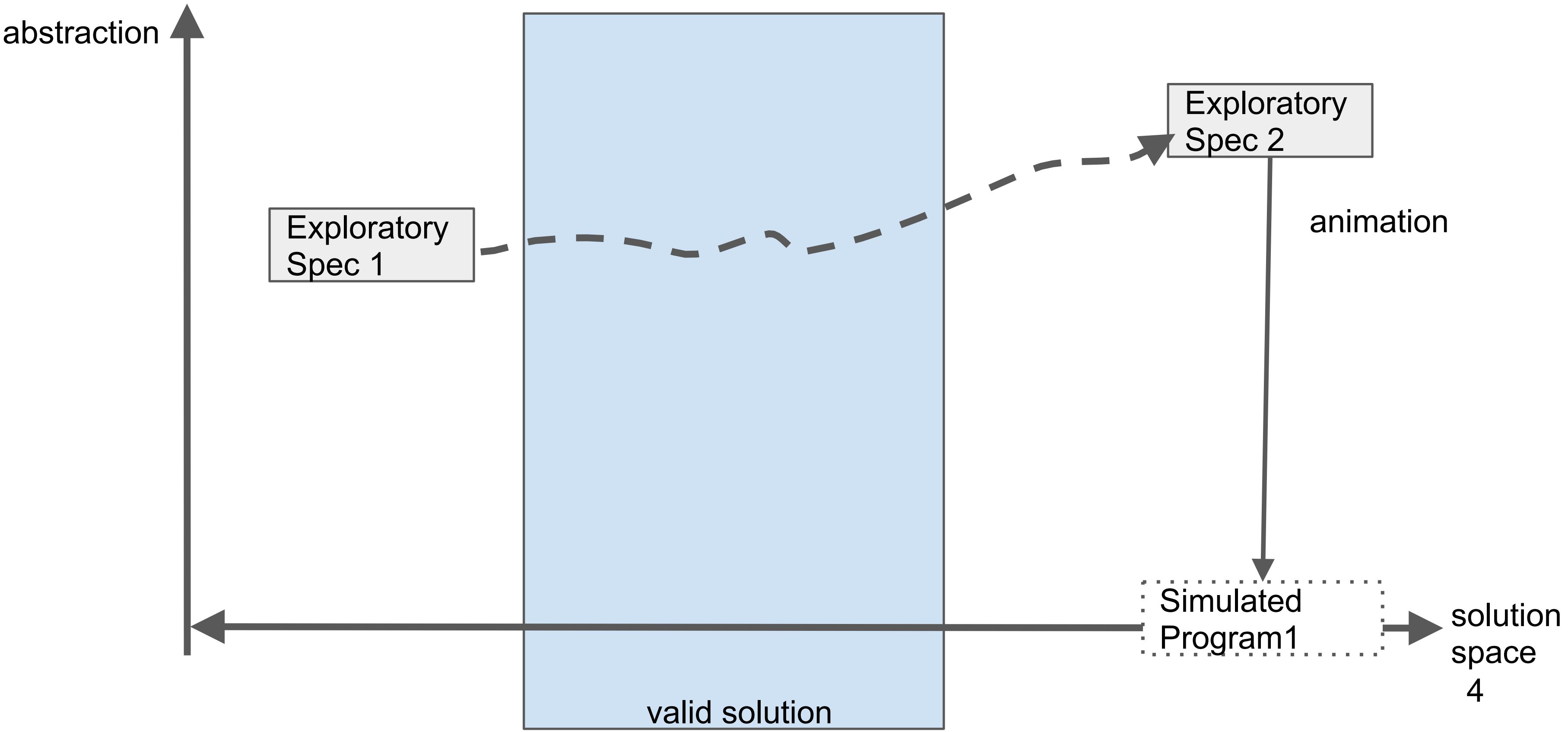
Agenda

1. Introduction: Exploratory Modeling
2. Code Generator for Exploratory Modeling
3. ViennaTalk and its Code Generators (Demo)
4. Performance and Discussion
5. Conclusion

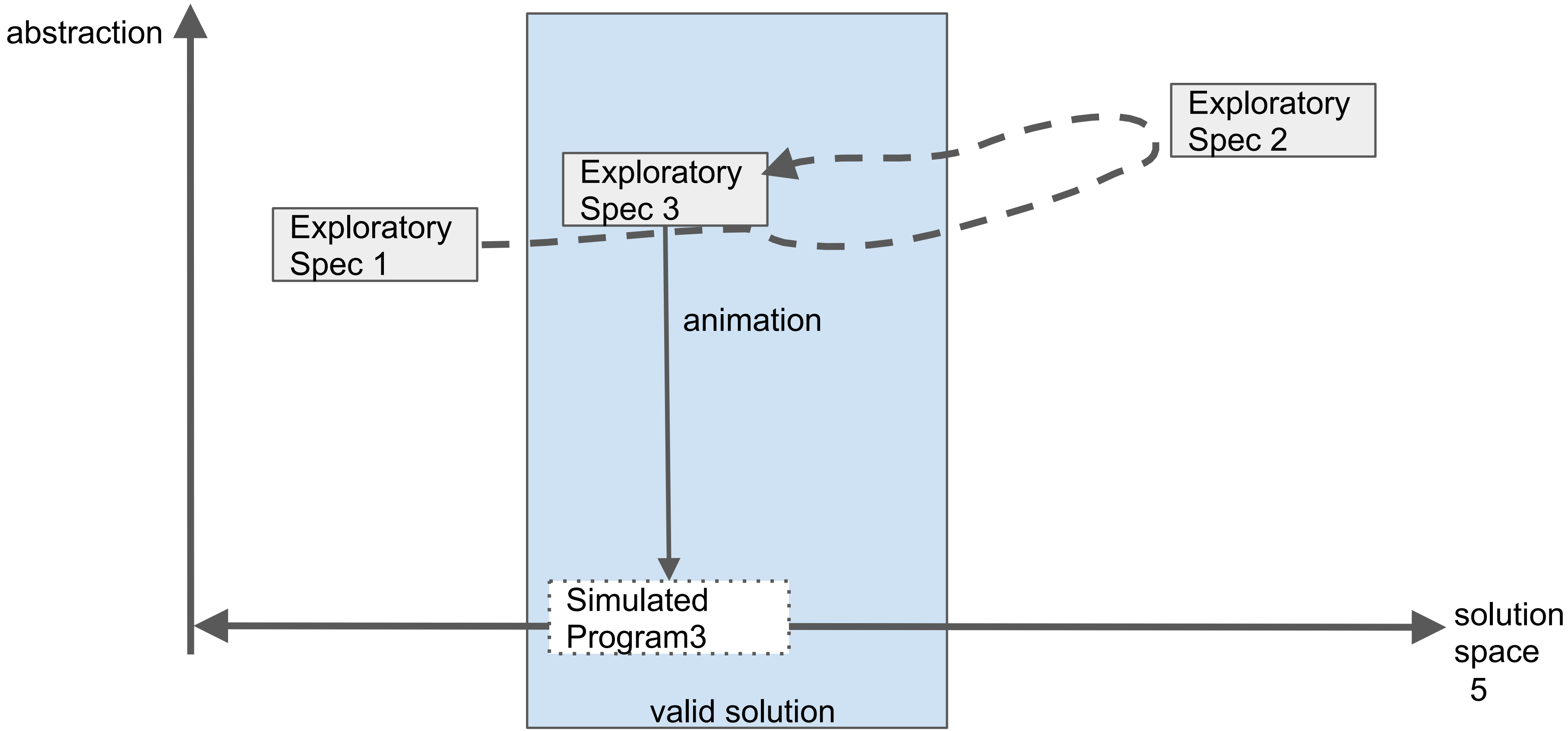
exploratory specification using animation



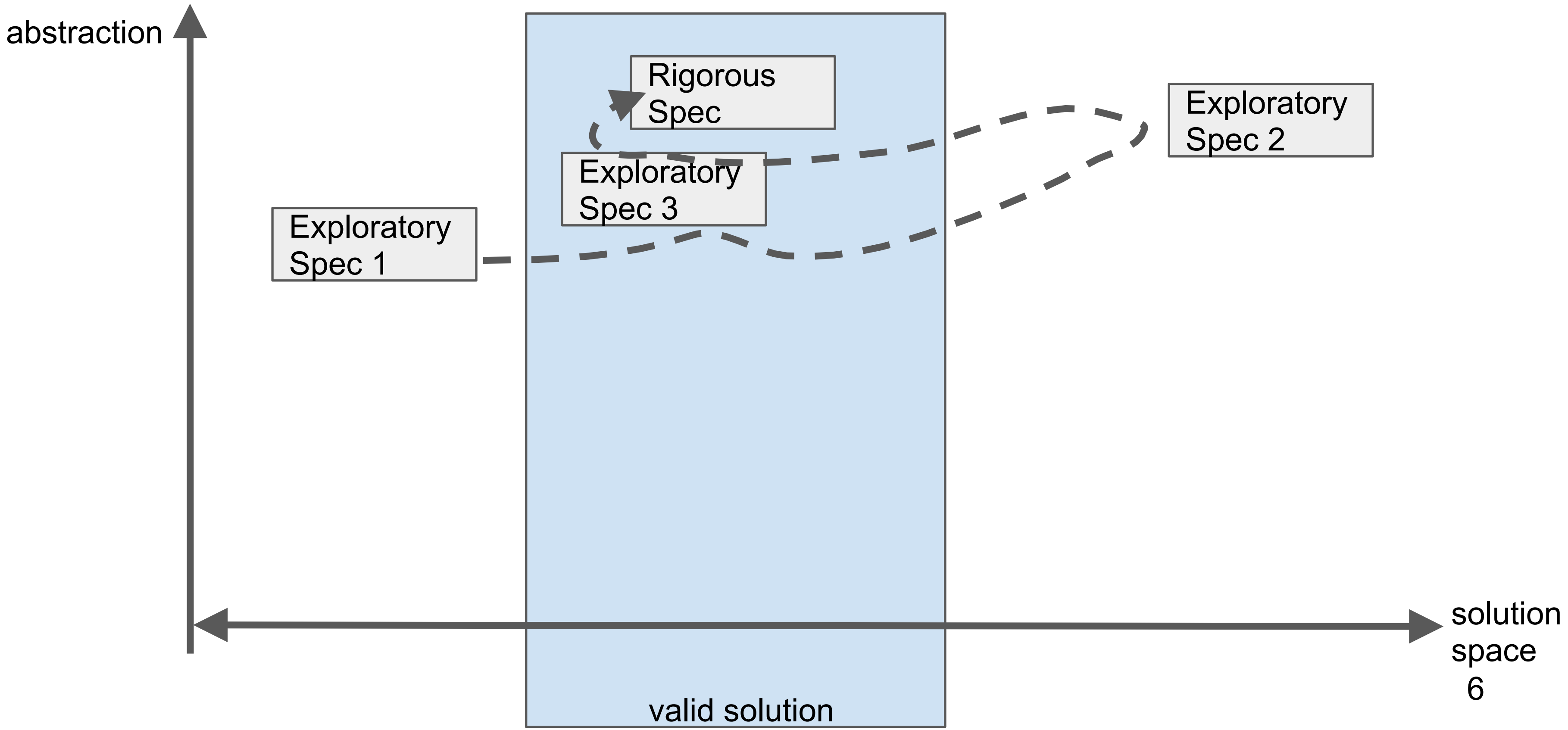
exploratory specification using animation



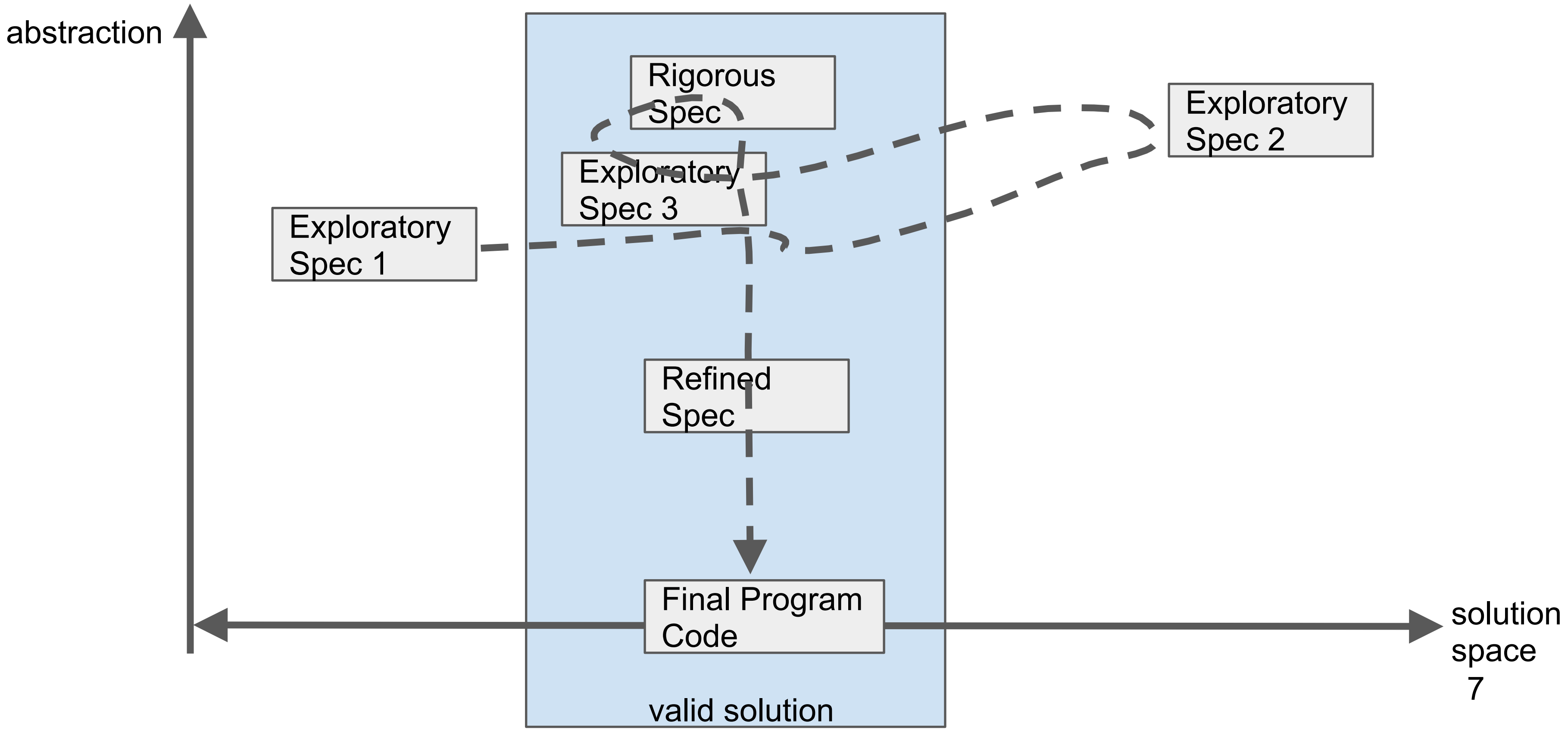
exploratory specification using animation



rigorous specification



design and implementation

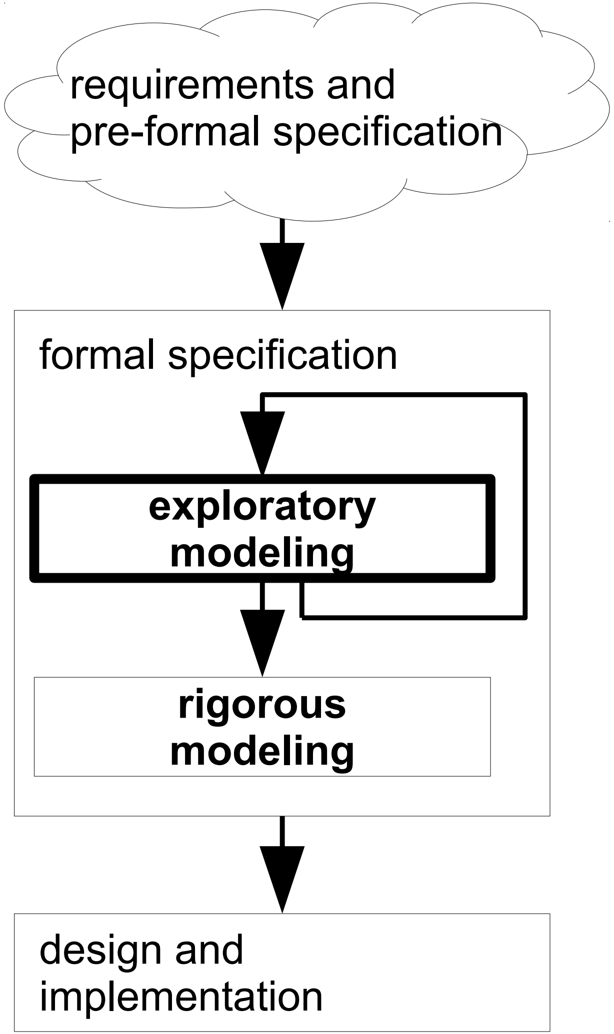


Exploratory Modeling

At the beginning of a development,

- specification is **incomplete**
- domain knowledge is **fragmented**
- **limited shared understanding** among stakeholders

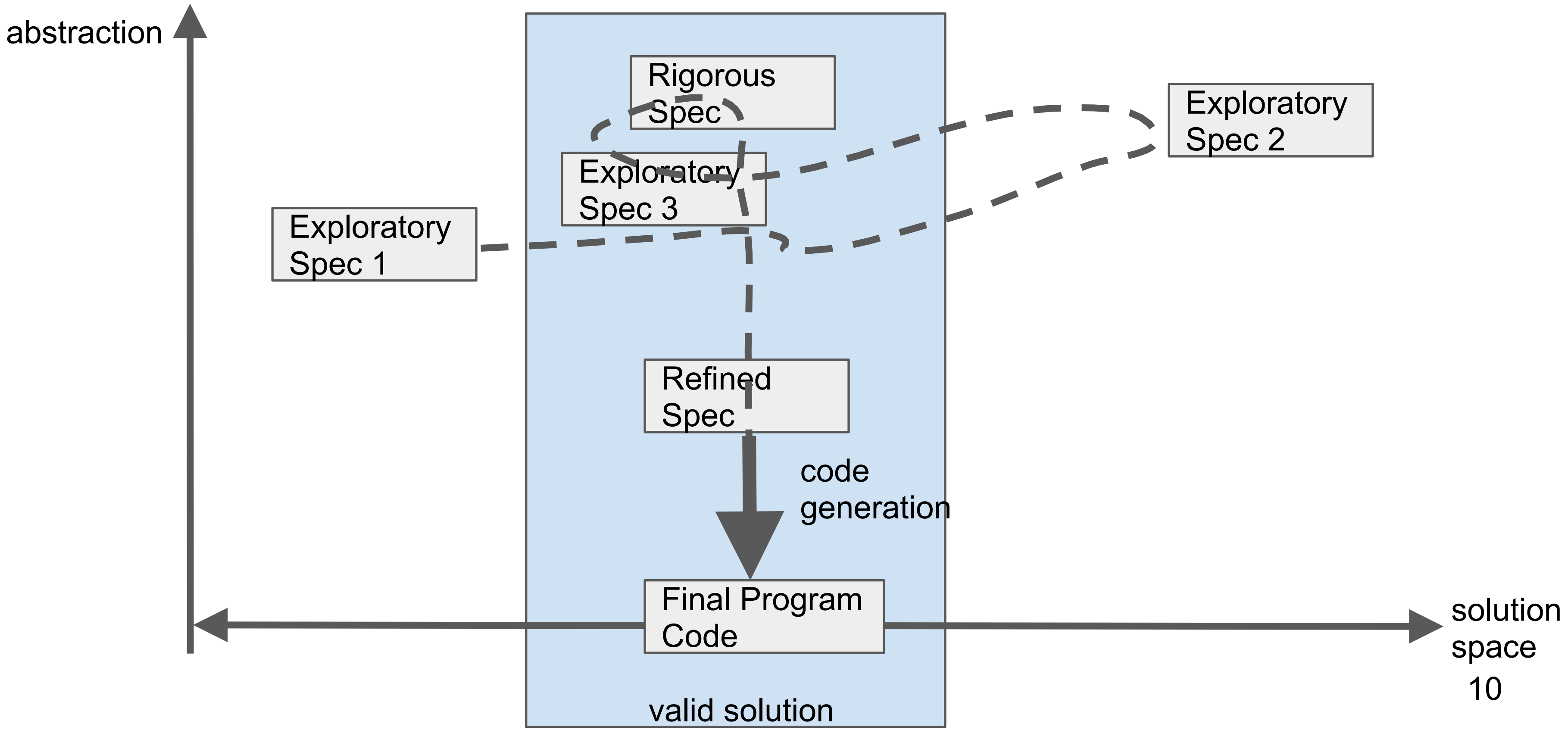
We learn through exploration over the problem domain by writing/analyzing/animating incomplete specifications



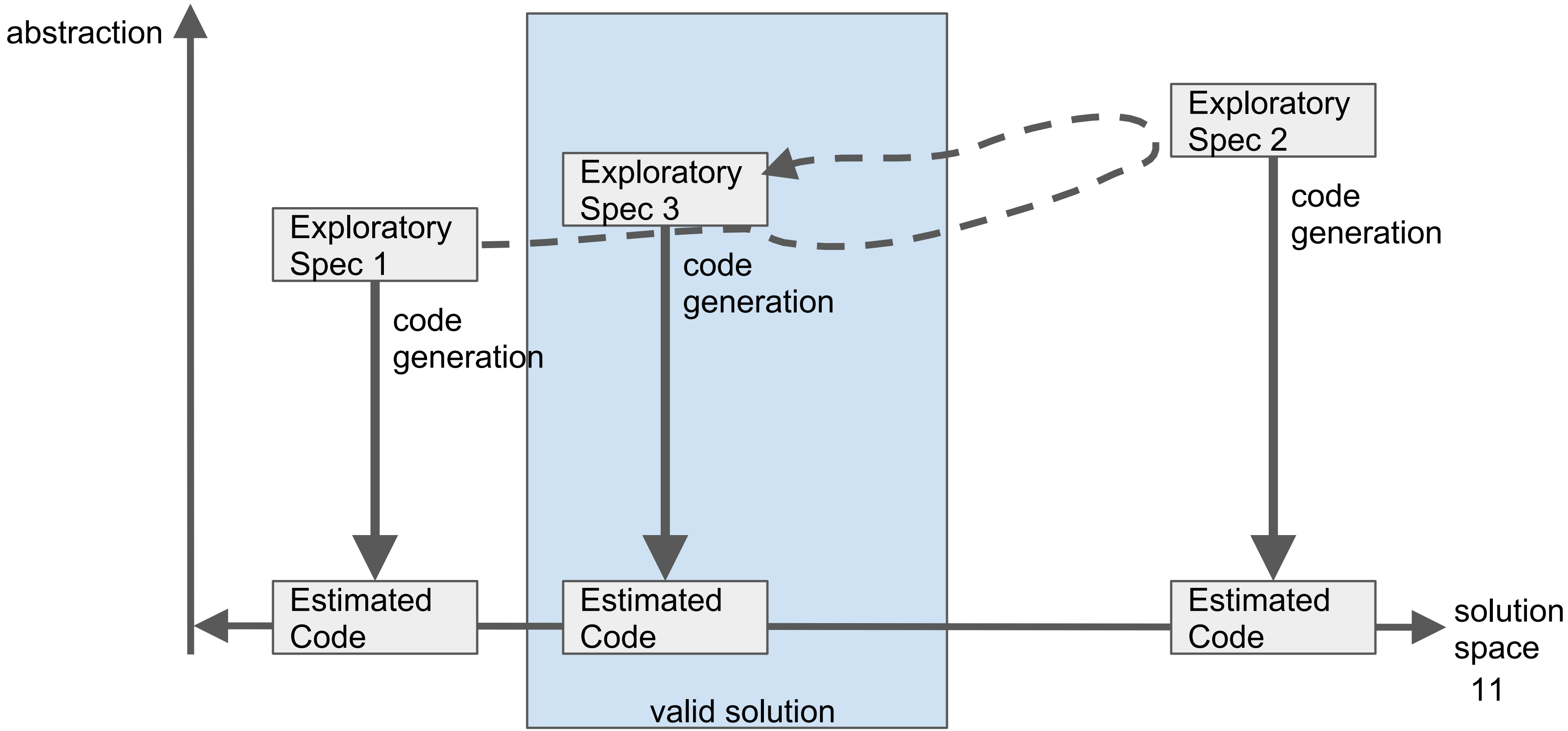
Code Generator

When do we use code generators?

Conventional use of Code Generator: Automated Implementation



Exploratory use of Code Generator: Animation by transpiler instead of interpreter



Transpiler vs Interpreter

Pros

- Performance
- Can modify generated code
- Linking and testing with existing implementation

Cons

- Debug (step execution, source level debugging)

Requirements to code generators for exploration (1/2)

Automatic compilation and execution of the generated source code
because you'll run it often.

Compilation and execution of the generated source code must be
controlled **in the same IDE** as the exploratory modeling
because it should be "integrated" in the environment.

Few limitations on the specification language for automated code
generation.
Don't bend the specification!

Requirements to code generators for exploration (2/2)

Debug capability must be enabled

because the specification is incomplete.

Understandable by stakeholders with no formal methods background

Use GUI builders or visualization tools

Permissive checking by choice

Continuous analysis

because the specification is incomplete.

ViennaTalk's code generators

- VDM-SL to **Smalltalk**
- **generate, compile, and run** smalltalk code **by just one click**,
- able to **turn on/off** type **checking** and assertion checking,
- use classes and objects in **Smalltalk's standard library** so that it looks like a hand coded Smalltalk program,
- generate **human readable/modifiable** code, and
- generate a **script, a class library or objects** from VDM-SL spec

Demo

Tic-Tac-Toe

Performance benchmark: Prime numbers

```
state Eratosthenes of
    space : seq of nat1
    primes : seq of nat1
init s == s = mk_Eratosthenes([], [])
end

operations
    setup : nat1 ==> ()
    setup(x) ==
        (space := [k | k in set {2, ..., x}];
        primes := []);
```

```
operations
  next : () ==> [nat1]
  next() ==
    cases space:
      [x] ^ - ->
        (primes := primes ^ [x];
        sieve(x);
        return x),
      others -> return nil
      end;

  sieve : nat1 ==> ()
  sieve(x) ==
    space := [space(i) | i in set inds space & space(i) mod x <> 0];
```

```
operations
  prime10000 : () ==> seq of nat1

  prime10000() ==
    (setup(10000);
    while next() <> nil do skip;
    return primes);
```

Performance of VienntaTalk's code generator

Tool	Interp/CG	Impl Lang	Time (Overture CG=1)
VDMTools	Intp	C++	79.6
VDMJ	Intp	Java	19.1
ViennaTalk	CG	Smalltalk(Script)	3.45
VDMTools	CG	C++	1.22
Overture tool	CG	Java	1.00
ViennaTalk	CG	Smalltalk(Class)	0.729
ViennaTalk	CG	Smalltalk(Object)	0.700

Performance of ViennaTalk (**with Pharo5**)'s CG

Tool	Interp/CG	Impl Lang	Time (Overture CG=1)
VDMTools	Intp	C++	79.6
VDMJ	Intp	Java	19.1
ViennaTalk	CG	Smalltalk(Script)	**2.26** (was 3.45 w/ Pharo4)
VDMTools	CG	C++	1.22
Overture tool	CG	Java	1.00
ViennaTalk	CG	Smalltalk(Class)	**0.639** (was 0.729 w/ Pharo4)
ViennaTalk	CG	Smalltalk(Object)	**0.608** (was 0.700 w/ Pharo4)

Why Smalltalk often run faster than C++/Java?

In general, Smalltalk is 2~10 time slower than Java

if they run the same algorithm.

...but

Example: Prime number

```
functions
    isPrime : nat1 -> bool
    isPrime(x) ==
        x <> 1 and (forall y in set {2, ..., x - 1} & x mod y <> 0);
```

what will happen if you evaluate **isPrime(20161107)**?

Example: Prime number

```
functions
    isPrime : nat1 -> bool
    isPrime(x) ==
        x <> 1 and (forall y in set {2, ..., x - 1} & x mod y <> 0);
```

what will happen if you evalu Prime(20161107)?

> This set eats really big memory!

Smalltalk code from

```
forall y in set {2, ..., x - 1} & x mod y <> 0
```

```
[:_forall | _forall allSatisfy: [ :y | x ¥¥ y ~= 0 ] ]
        value: (2 to: x - 1)
```

This **interval** object holds only two integers.

Why Smalltalk often run faster than C++/Java?

VDM specs defines the problem, not the algorithm.

A language with **high abstraction and rich library** provides

- a **good selection of algorithms** and
- their **fine implementations**.

Using objects and classes from the standard library naturally brings the above benefits to the generated code.

Summary

Code generators for exploratory modeling

- Shall be **handy to use**… **all-in-one reflective environment**
- Shall have **less limitation** to the spec … as less as interpreters
- Shall do **flexible checking** … as flexible as interpreters
- Shall run **fast** … with **rich and abstract** language/libraries

Future work

- Shall **debug** the VDM spec … step executions with VDM-SL source
- Shall support **testing**