Specification of Chemical Formulæ in XL with Operator Overloading

Reinhard Hemmerling

University of Göttingen

28 February 2012

Outline

Motivation

- ultimately we want to specify reactions like

$$2\,\mathrm{H_2} + \mathrm{O_2} \underset{k_b}{\overset{k_f}{\rightleftharpoons}} 2\,\mathrm{H_2O}$$

- use **operator overloading** to enable this in XL
- automatically derive **differential equations** from this and solve these numerically
- possible for **elementary reactions** via **law of mass action**

Operator Functions

- like in C++
- define function with **special name**
- name is `operator<symbol>`, where `<symbol>` is one of `+`, `-`, `*`, `/`, `%`, ...
- unary vs. binary operators
- count parameters of function
- plus one extra for non-static functions (left operand is `this`)

Operator Functions — Example

```
class Complex {
  double real, imag;

  public Complex(double real, double imag) {
    this.real = real;
    this.imag = imag;
  }

  public static Complex operator+ (Complex a, Complex b) {
    return new Complex(a.real + b.real, a.imag + b.imag);
  }

  ...
}
```

Operator Functions — Conclusion

- allows:

```
Complex a = new Complex(1, 2);
Complex b = new Complex(3, 4);
Complex c = a + b;
```

- but not yet:

```
Complex d = 1 + a;
Complex e = a + 1;
```

- could be resolved by providing additional operator overloads
- better solutions are **implicit conversions**

Implicit Conversion

- like in C++
- extends autoboxing from primitive to reference types
- can be defined in any of the following ways:

```
class C {
  C(S source);
  static C valueOf(S source);
  T toT();
  t tValue();
}
```

- follows existing patterns like `toString()`, `intValue()`, etc.

Implicit Conversion — Example

```
class Complex {
  ...

  @de.grogra.xl.lang.ConversionConstructor
  public C(double d) {
    real = d;
    imag = 0;
  }
}
```

Chemical Kinetics — Introduction

- *reaction equation*:

$$\nu_1 \mathrm{A} + \nu_2 \mathrm{B} + \ldots \longrightarrow \nu_1' \mathrm{A}' + \nu_2' \mathrm{B}' + \ldots$$

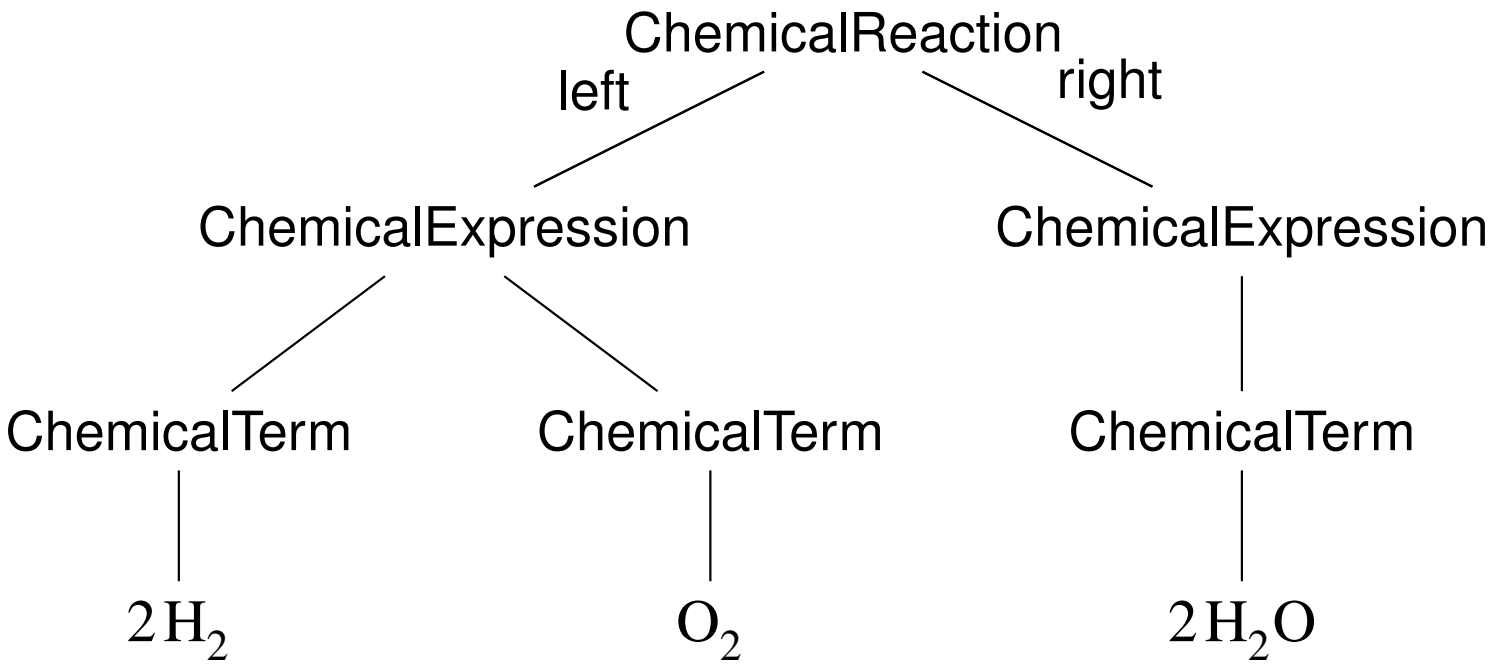
- *reaction rate*:

$$r = -\frac{1}{\nu_1}\frac{d[\mathrm{A}]}{dt} = -\frac{1}{\nu_2}\frac{d[\mathrm{B}]}{dt} = \frac{1}{\nu_1'}\frac{d[\mathrm{A}']}{dt} = \frac{1}{\nu_2'}\frac{d[\mathrm{B}']}{dt} = \ldots$$

- *rate law*:

$$r = k[\mathrm{A}]^{\nu_1}[\mathrm{B}]^{\nu_2}\ldots$$

Chemical Kinetics — Parse Tree Generation



---

in XL: `2*H2 + O2 <=> 2*H2O;`

Chemical Kinetics — Molecule

```
// each entity in the reaction is a molecule
class Molecule {

  // predefine some instances of Molecule so that they
  // can be statically imported into current scope
  public static final Molecule H2  = new Molecule("H2");
  public static final Molecule H2O = new Molecule("H2O");
  public static final Molecule O2  = new Molecule("O2");

  String name;

  public Molecule(String name) {
    this.name = name;
  }

  public String toString() {
    return name;
  }
}
```

Chemical Kinetics — Reaction Arrow

```
// stores the whole reaction
class ChemicalReaction {
  ChemicalExpression left, right;
  double kf, kb;
}

// infer reaction from left and right side
public static ChemicalReaction operator<=> (
  ChemicalExpression lhs, ChemicalExpression rhs)
{
  ChemicalReaction result = new ChemicalReaction ();
  result.left = lhs;
  result.right = rhs;
  return result;
}
```

Chemical Kinetics — Chemical Expressions

```
// an expression is a list of terms
class ChemicalExpression {
  final ArrayList<ChemicalTerm> terms =
    new ArrayList<ChemicalTerm>();

  public void add(ChemicalTerm term) {
    terms.add(term);
  }
}

// terms are combined by + operator
public static ChemicalExpression operator+ (
  ChemicalExpression expr, ChemicalTerm term)
{
  expr.add(term);
  return expr;
}
```

Chemical Kinetics — Chemical Term

```
// factor is stoichiometric coefficient for molecule
class ChemicalTerm {
  double factor;
  Molecule m;

  public ChemicalTerm(Molecule m) {
    this(1, m);
  }

  public ChemicalTerm(double factor, Molecule m) {
    this.factor = factor;
    this.m = m;
  }
}

public static ChemicalTerm operator* (
  double factor, ChemicalTerm term)
{
  term.factor *= factor;
  return term;
}
```

Chemical Kinetics — Implicit Conversions

Some implicit conversions are needed to cover remaining cases:

1) `Molecule` → `ChemicalTerm`
2) `Molecule` → `ChemicalExpression`
3) `ChemicalTerm` → `ChemicalExpression`

Chemical Kinetics — Application

- put all reactions into a `Model`
- derive ODEs from reaction equation for numerical integration
- use law of mass action
- need to assign indices to each `Molecule`
- map `Molecule` $\rightarrow$ `int` with `java.util.HashMap`
- can then use any standard method for integration

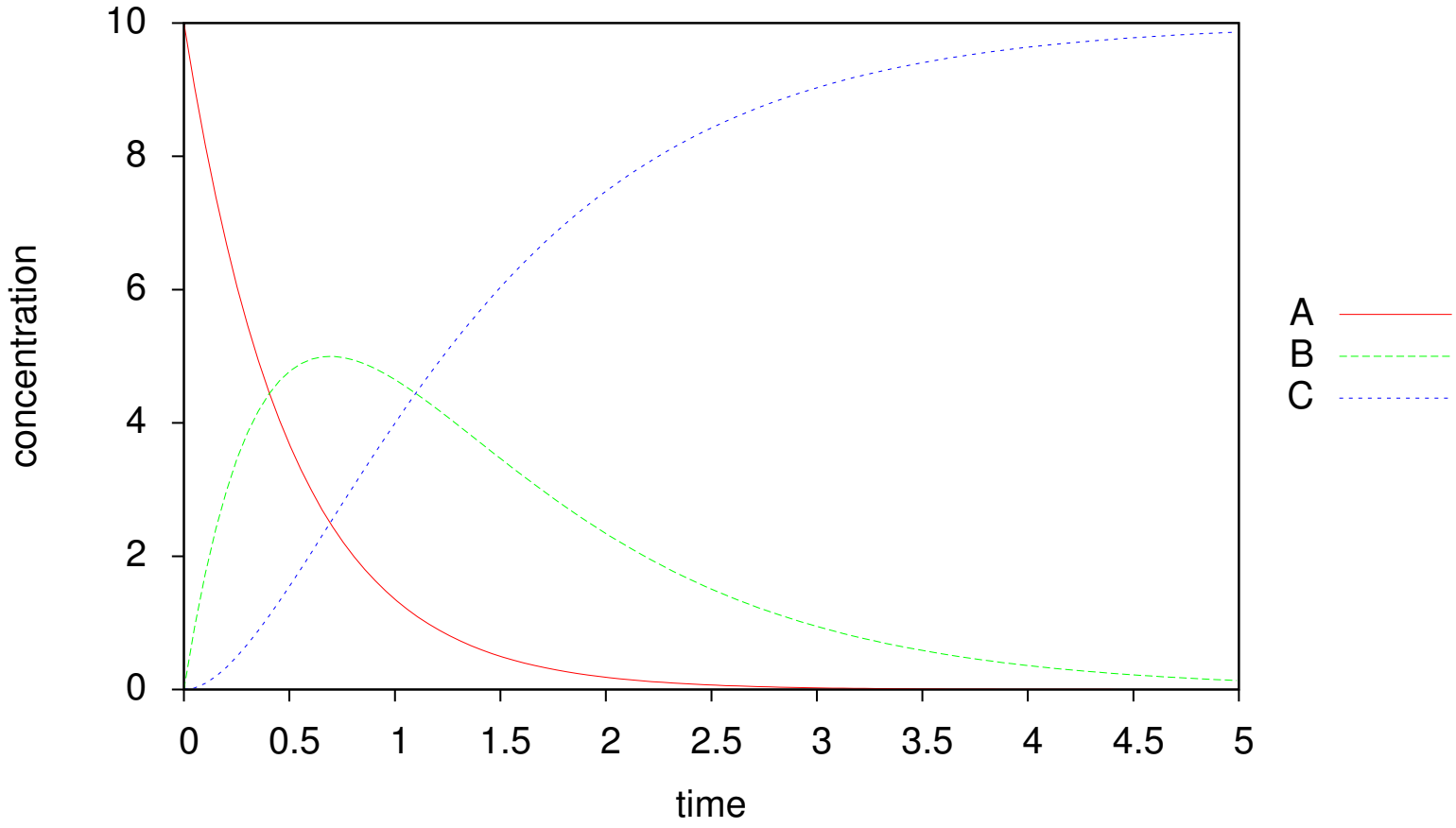
Example 1 — Reaction

$$\mathrm{A} \xrightarrow{k_1} \mathrm{B} \xrightarrow{k_2} \mathrm{C}$$

```
const Molecule A = new Molecule("A");
const Molecule B = new Molecule("B");
const Molecule C = new Molecule("C");

public void run()
{
  ChemicalReaction r1 = A <=> B;   // first reaction: A -> B
  r1.setForwardRateConstant(2);

  final Model model = new Model();
  model.addSlope(r1);
  model.add(B <=> C, 1);           // second reaction: B -> C

  ...
```

Example 1 — Reaction

$$\mathrm{A} \xrightarrow{k_1} \mathrm{B} \xrightarrow{k_2} \mathrm{C}$$



Example 2 — Michaelis-Menten Kinetics

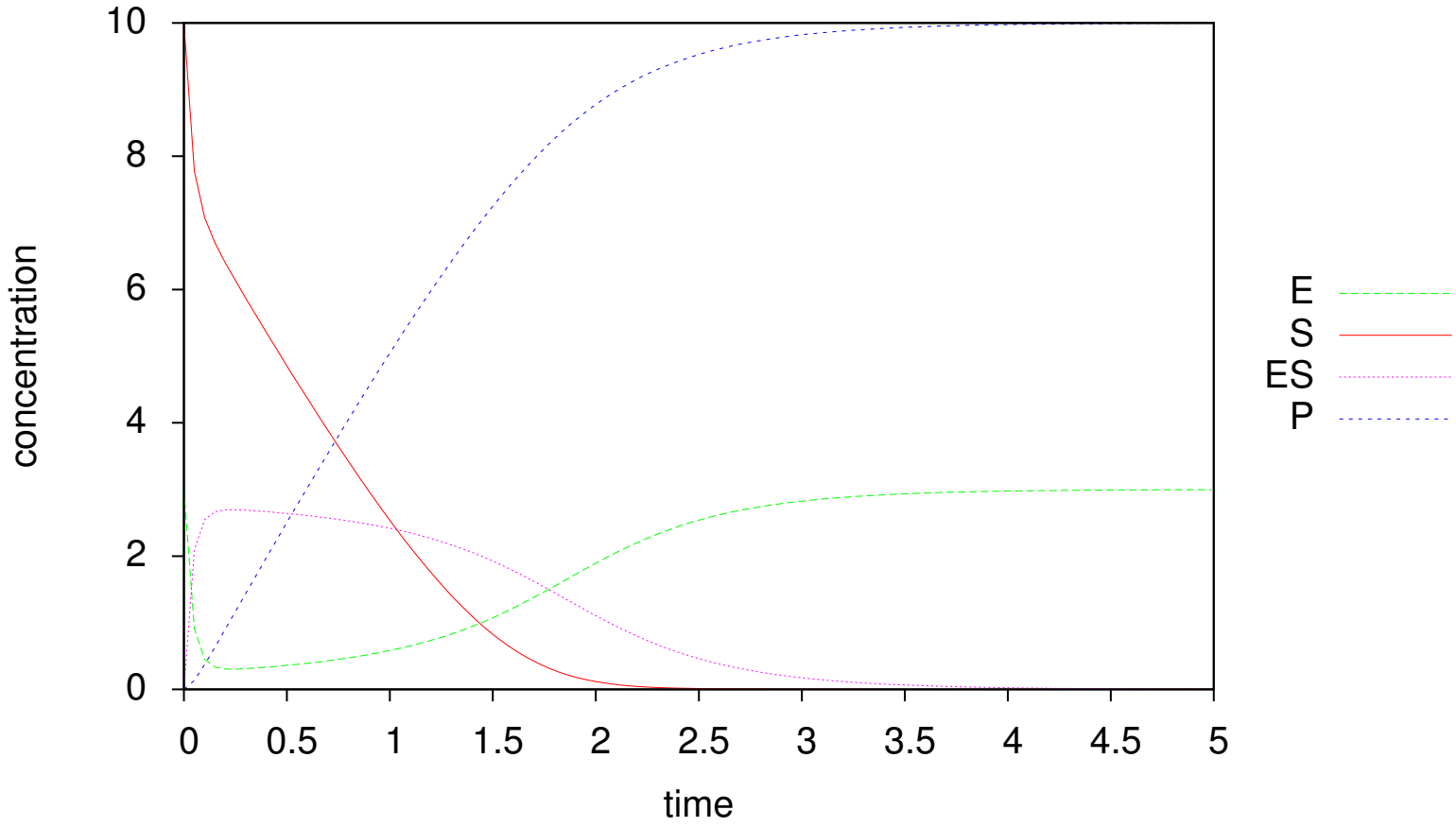
- reaction:

$$\mathrm{E} + \mathrm{S} \underset{k_r}{\overset{k_f}{\rightleftharpoons}} \mathrm{ES} \xrightarrow{k_{cat}} \mathrm{E} + \mathrm{P}$$

- in XL:

```
final Model model = new Model();
model.add(E + S <=> ES, 3, 0.1);
model.add(ES <=> E + P, 2);
```

Example 2 — Michaelis-Menten Kinetics



Summary

- operator overloading very versatile
- makes chemical reactions part of the language
- syntax checking included
- also other applications
  - right-hand sides of rules
  - iostreams-like input/output
  - retrofit existing classes like `BigInteger` and `BigDecimal` with overloaded operators

Thank you

for your attention.

Backup Slides

Example 1 — Reaction

$$\mathrm{A} \xrightarrow{k_1} \mathrm{B} \xrightarrow{k_2} \mathrm{C}$$

```
...

// assign indices
final HashMap m = new HashMap();
int count = model.assignIndices(0, m);

// allocate memory
double[] y0 = new double[count];
double[] y  = new double[count];

// set initial conditions
setValue(m, y0, A, 10);

...
```

Example 1 — Reaction

$$\mathrm{A} \xrightarrow{k_1} \mathrm{B} \xrightarrow{k_2} \mathrm{C}$$

```
...

// prepare differential equations
// getRate() will be called by the integrator
ODE ode = new ODE()
{
  public void getRate(double[] out,
    double t, double[] y)
  {
    Arrays.fill(out, 0);
    model.eval(out, t, y);
  }
};

...
```

Example 1 — Reaction

$$\mathrm{A} \xrightarrow{k_1} \mathrm{B} \xrightarrow{k_2} \mathrm{C}$$

```
...

// create numerical solver
Solver solver = new FirstOrderIntegratorAdapter(
  new ClassicalRungeKuttaIntegrator(0.001));

...
```

Example 1 — Reaction

$$\mathrm{A} \xrightarrow{k_1} \mathrm{B} \xrightarrow{k_2} \mathrm{C}$$

```
...

// setup monitor function to plot state over time
solver.setMonitor(1, new Monitor()
{
  public void g(double[] out, double t, double[] y)
  {
    // trigger at 20Hz
    out[0] = sin(PI * t * 20);
  }
  public boolean handleEvent(int i, double t, double[] y)
  {
    // plot data or whatever
    return false;
  }
});

...
```

Example 1 — Reaction

$$\mathrm{A} \xrightarrow{k_1} \mathrm{B} \xrightarrow{k_2} \mathrm{C}$$

```
    ...

    // perform integration
    solver.integrate(ode, 0, y0, 5, y);
}
```