Poplar

Extending the Java Programming Language for Evolvable Component Integration

Johan T. Nyström Persson
Honiden Laboratory
Department of Computer Science, University of Tokyo

January 16, 2012

Thesis supervisor: Shinichi Honiden
Head of thesis committee: Masami Hagiya

Outline

- Introduction
- Design
- Demo
- Formalisation
- Implementation
- Case study and evaluation
- Conclusion

Java components

- Component-based and object-oriented software are now dominant paradigms
- Java is an extremely successful OO language
- However, essential difficulties remain in **integrating** components and **preserving integrations**
    - In this work: component = set of Java classes

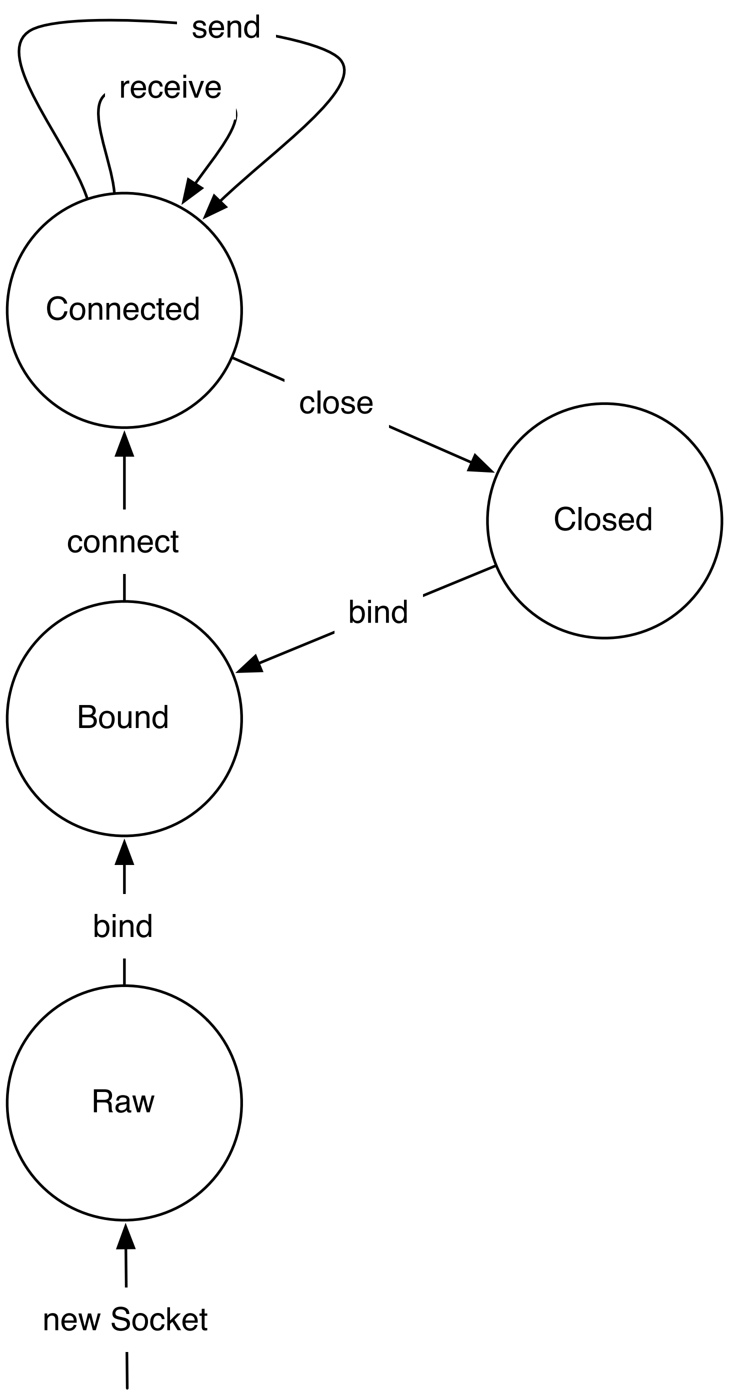
Depending on a component

```
public class Socket {
    Socket() {}
    void bind(SocketAddress bindPoint) {}
    void connect() {}
    void send(byte[] data) {}
    void receive(byte[] data, int offset, int max)
{}
    void close () {}
}
```

```
class Client {
    void m(SocketAddress sa) {
        Socket s = new Socket();
        s.bind(sa);
        s.connect();
        s.send(data);
        s.close();
    }
}
```

- **What knowledge does a programmer need** in order to make use of this Socket API?
- **What must remain unchanged in the future**, in order for a client class to recompile correctly?

Temporal assumptions (protocol[1]/typestate[2])




```
public class Socket {
        Socket() {}
        void bind(SocketAddress bindPoint) {}
        void connect() {}
        void send(byte[] data) {}
        void receive(byte[] data, int offset, int max) {}
        void close () {}
}
```

- In general, methods of objects cannot be called at any time. Sequential constraints apply.
- We may be assuming that calling **send** is valid once we have called **connect**, and so on

1. Yellin, Daniel M and Strom, Robert E. *Protocol Specifications and Component Adaptors*. TOPLAS, 1997.
2. Deline, R. and Fähndrich, M. *Typestates for Objects*. ECOOP 2004.

Monday, 16 January 2012

Note that without typestate, there is no way to statically check that clients use temporal constraints correctly

Semantic assumptions (method contract)

```
public class Socket {
        Socket() {}
        void bind(SocketAddress bindPoint) {}
        void connect() {}
        void send(byte[] data) {}
        void receive(byte[] data, int offset, int max) {}
        void close () {}
}
```

- For each possible combination of input arguments, does the method have a well defined behaviour?
- We expect that...
  - Data passed to the **send** method will be sent to the connected remote socket
  - **receive** stores data in the array passed as the first parameter
  - No files will be deleted from the hard drive (etc.)

Dependencies

- **Constraints on future versions** of service components
  - Temporal constraints must not be strengthened, only weakened.
  - For semantic contracts, the **substitution principle**[1] must be valid
    - Only weaker preconditions or stronger postconditions are acceptable changes
  - Syntactic/structural changes, such as renaming or incompatible refactoring, are unacceptable

1. Liskov, B. and Wing, J. *A Behavioural Notion of Subtyping*. TOPLAS 1994

The problem

- **Essential conflict between evolution and composition**
  - Components need to evolve post-deployment[1]
  - **Evolution of semantic contracts/ temporal contracts may be necessary but this threatens integration**
    - A lot of manual work becomes necessary
  - "Procedure calls are the assembly language of software interconnection"[2]

1. Dig, D. and Johnson, R. *The Role of Refactorings in API Evolution.* ICSM 2005.
2. Shaw, M. *Procedure Calls are the Assembly Language of Software Interconnection.* 1993.

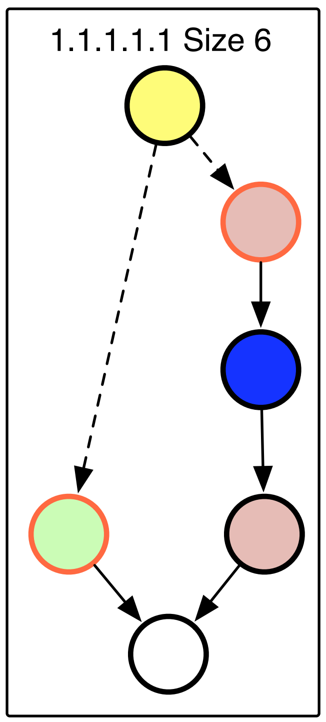
Key concept

- Instead of relying on “the assembly language of procedure interconnections”, **generate integration code automatically!**
- **Re-generate after evolution**
- Specify integrations with a **minimum of information** so that the chance of finding a solution is high

Related work

- AI planning
- Typestate and protocols
- Labelled argument selection
- Prospector (“Jungloid mining”)
- Effect systems

Related work: AI planning[1]



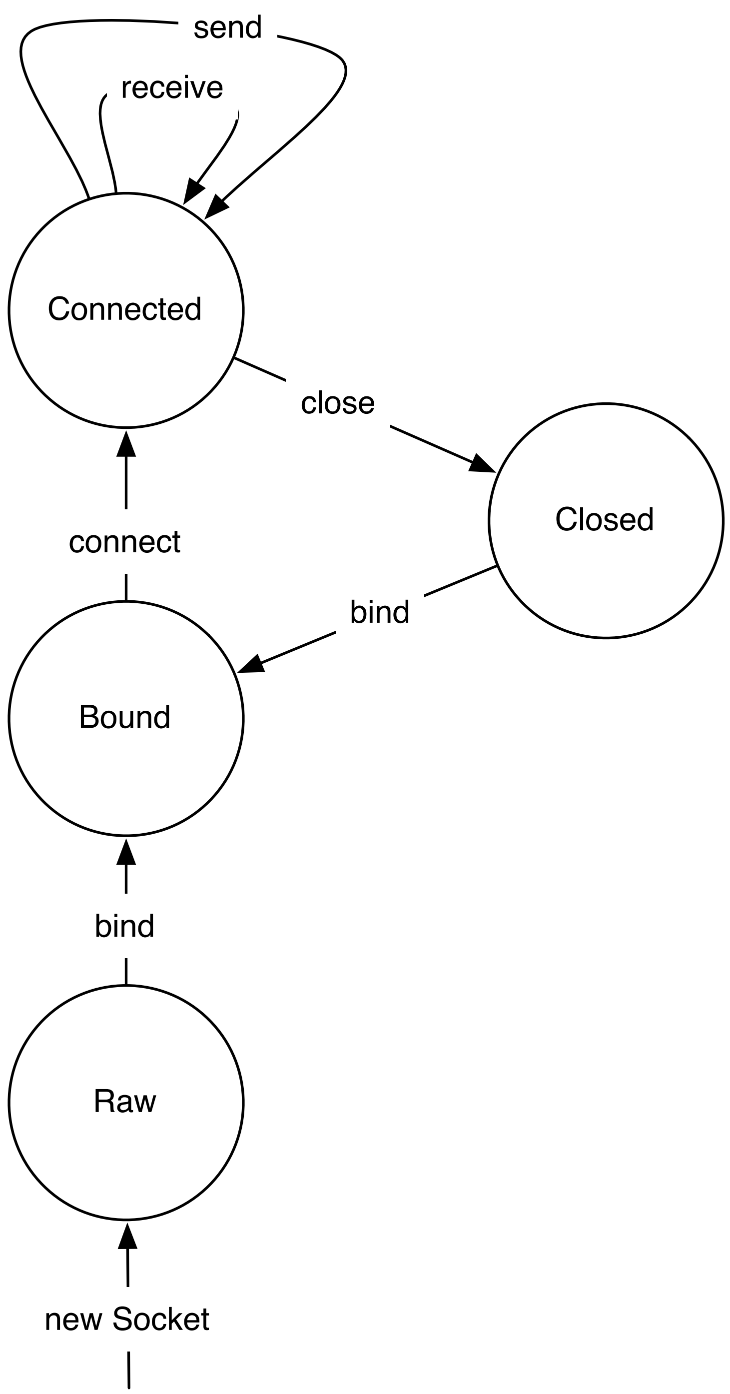
- AI planning is the problem of assembling a sequence of actions to convert an initial state to a goal state
- Intuitively, this is very close to the problem of constructing valid API usage patterns
  - It also seems to resemble what programmers must do manually...
- How can we describe the domain so as to generate meaningful, safe Java fragments using AI planning?

1. Ghallab, Nau and Traverso. *Automated Planning: Theory & Practice*. 2004.

Approach

- Find a way to **describe Java code as an AI planning domain**, in such a way that the results make sense and are useful
- Borrow ideas from many well-studied fields to constrain and inform planning
- Use **simple techniques** to demonstrate the overall **proof of concept**

Related: protocol and typestate systems



- Well studied domain since the 1980's, especially popular for OO languages in the last 10 years[1,2,3]
- Typestate analysis constrains API clients to use valid sequences only
- **Use typestate to constrain AI planning**

1. Strom, R.E. and Yemini, S. *Typestate: A Prog. Lang. Concept for Enhancing Software Reliability*. TSE 1986
2. Deline, R. and Fähndrich, M. *Typestates for Objects*. ECOOP 2004
3. Bierhoff, Kevin and Aldrich, Jonathan. *Modular Typestate Checking of Aliased Objects*. OOPSLA 2007.

Monday, 16 January 2012

Note that without typestate, there is no way to statically check that clients use temporal constraints correctly

Related: Prospector

- Prospector[1] is an interactive tool that constructs code fragments by matching argument types with return types
- A valid codebase is mined in advance to extract patterns
  - Patterns are composed according to **type compatibility**
- **User requests a type to be generated in a context**
  - Borrow this idea, but avoid the need to mine a codebase

1. Mandelin, D., Xu, L., Kimelman, D., and Bodik, R. *Jungloid Mining: Helping to Navigate the API Jungle.* PLDI 2005.

Related: labelled argument selection

- Some languages (e.g. Lisp, ADA) allow for argument reordering and omission based on labels
- Labelled lambda calculus[1,2] allows for automatic argument selection from a set based on labels
  - This is more powerful than Prospector, which *only* uses type information
  - What if we use both types and labels to select?

1. Garrigue, Jacques. *Label-Selective Lambda Calculi and Transformation Calculi.* 1994
2. Garrigue, Jacques and Ait-Kaci, Hassan. *The Typed Polymorphic Label-Selective Lambda-Calculus.* POPL 1994.

Optionally cut from main section

Related: effect systems

- Effect systems are a well studied class of type systems that **annotate terms with their side effects**
- For OO languages, systems that reason about heap reads and writes in terms of **polymorphic regions** have been well studied[1,2]
- **Use this idea to constrain AI planning and avoid unwanted interference**

1. Leino, K.R.M, Poetzsch-Heffter, A and Zhou, Y. *Using Data Groups to Specify and Check Side Effects*. PLDI 2002.
2. Greenhouse, A and Boyland, J. *An Object-Oriented Effect System*. ECOOP 1999

Optionally cut from main section

Hypothesis

"A combination of AI planning, labelled variables and temporal specifications, when applied to the Java programming language, can yield a fully automatic integration technique that is robust to evolution."

(Robust to evolution: gracefully handles cases that cannot be handled by standard Java, either finding a solution automatically or correctly reporting an error)

Note: even though we are investigating this approach in the context of Java, it should be straightforward to transfer it to other imperative OO languages. For instance, C#. Strong typing is good, reduces ambiguity, so I expect C++ will not be as easy.

Contribution

- A Java extension, Poplar
  - Fully automatic component integration using declarative specifications
    - Also: checking that methods conform to their contracts
  - Modular analysis and compilation
  - Formalisation, implementation, case study

Outline

- Introduction
- Design
- Demo
- Formalisation
- Implementation
- Case study and evaluation
- Conclusion

11 min mark

Example - socket client

V. 1

```
public class Socket {
        Socket() {}
        void connect() {}
}
```

```
public class Client {
    void m(SocketAddress a)
      a: remoteAddress. {
        Socket s = new Socket();
        s.connect();
    }
}
```

V. 2

```
public class Socket {
        Socket() {}
        void configure(boolean compress) {}
        void connect() {}
}
```

```
public class Client {
    void m(SocketAddress a)
      a: remoteAddress. {
        Socket s = new Socket();
        s.configure(false);
        s.connect();
    }
}
```

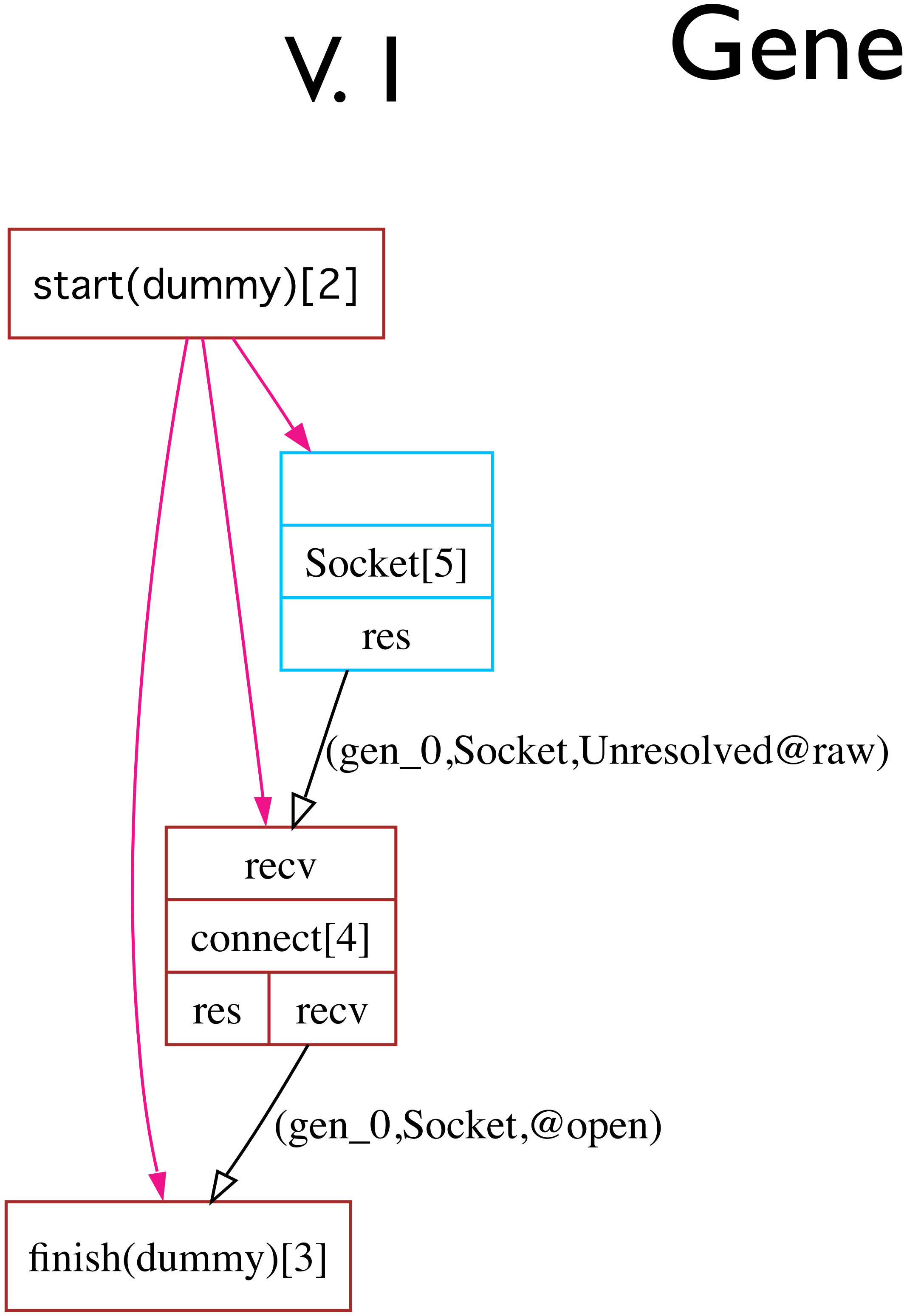
Annotate with labels and queries

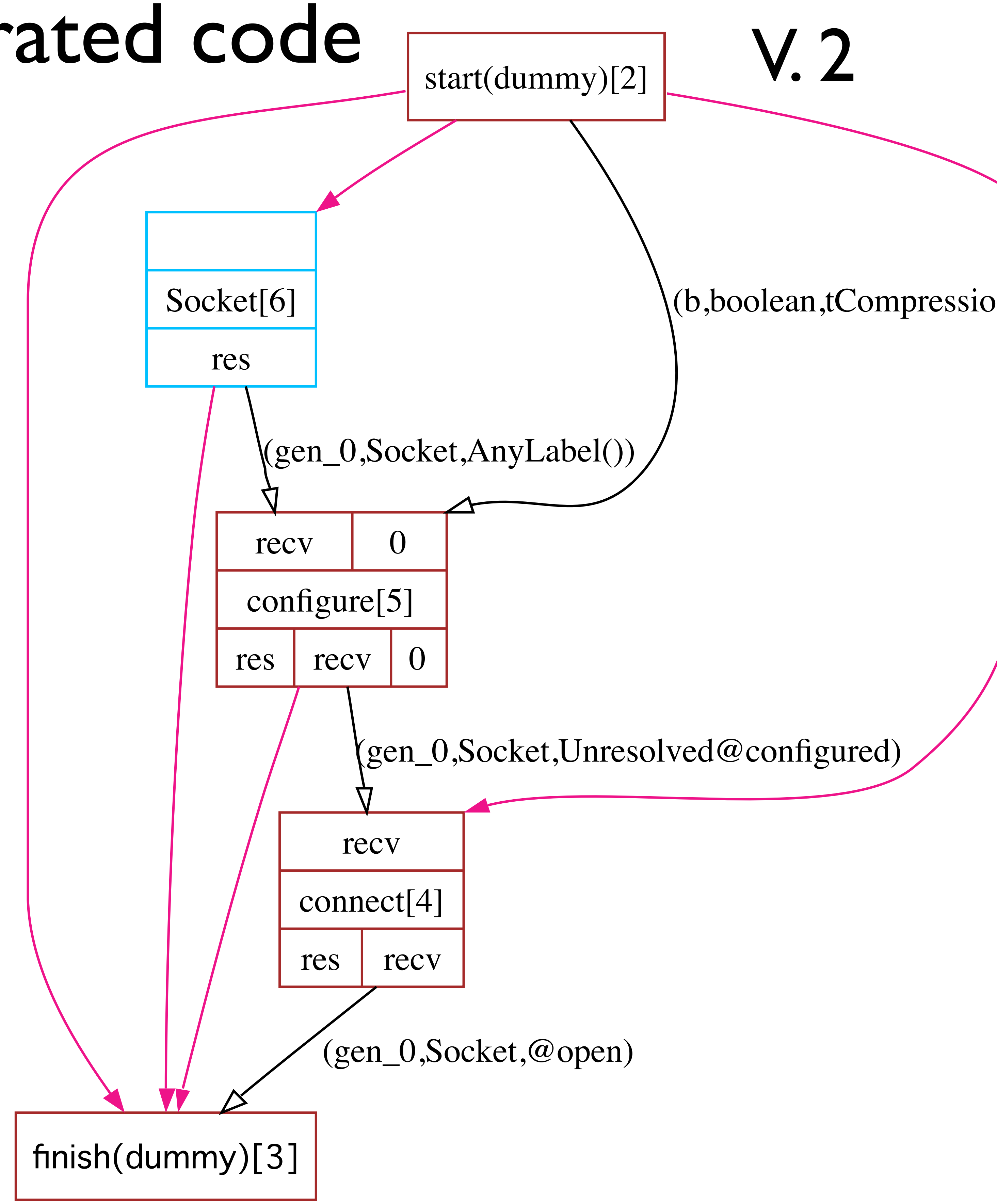
```
public class Socket {
    resource state {
       properties @open, @raw;

       Socket() this: ++@raw. { }
       void connect() this: ++@open. { }
    }
}
```

```
public class Socket {
    resource state {
       properties @open, @raw, @configured;

       Socket() this: ++@raw. { }
       void configure(boolean compr)
         compr: tCompression;
         this: ++@configured. { }
       void connect()
         this: @configured, ++@open. { }
    }
}
```

```
public class Client {
    void m() {
        boolean b:(tCompression) = false;
        Socket s = #produce(Socket, @open);
    }
}
```

Generated code





For both versions, we can generate correct integration code without changing the client at all.

Design overview

- **Labels/state names** from typestate, protocols, labelled lambda calculus
- **Queries** from Prospector
- **Resources** from Boyland/Greenhouse effect system
- **Uniqueness kinds** from typestate, effect systems, many others

Labels

- Most central element in the design
- **Multiple roles**
  - Protocol/temporal state
  - Internal semantic contract (predicate on object's private state)
  - External semantic contract (anything)
- **Two kinds: properties and tags**

Properties (generalised typestate)

```
public class Socket {
    resource state {
     properties @raw, @bound,
         @open, @closed;

     Socket()
         this: ++@raw. { ... }
     void bind(SocketAddress bindPoint)
         this: -@raw, ++@bound. { ... }
     void connect()
         this: -@bound, ++@open. { ... }
     //...
    }
}
```

-@x: precondition (lost)

++@x: postcondition (added)

- **Destructible** labels, defined for a class
- Essential in order to encode temporal constraints
- Prefixed with @
- Gives each object potentially 2^n "states" for *n* properties
- Associated with a *resource*

Tags

```
public class Socket {
    resource state {
     properties @raw, @bound,
         @open, @closed;
          //...

    void send(byte[] data)
    this: @open; data: ++sentData.{ ... }
     //...
    }
}
```

- **Non-destructible labels**
- For irreversible effects (e.g. sending data)
- For identifying constants

Monday, 16 January 2012

This is not an essential feature, but nice to have

Queries

- Purpose: express integration goals
- Two kinds
  - **Produce** - request a value of a given type with a set of labels
  - **Transform** - request additional labels for a given variable
- Idea from Prospector (which has an equivalent of **produce**)

Produce-queries

```
public class Socket {
    resource state {
       properties @raw, @bound, @open, @closed;

       Socket() this: ++@raw. { }
       void bind(SocketAddress bindPoint) this: -@raw, ++@bound;
         bindPoint:remoteAddress. { }
       void connect() this: -@bound, ++@open. { }
       void send(byte[] data) this: @open; data: ++sentData. { }
    }
}
```

Generate and substitute

```
public class Client {
    void m(SocketAddress a)
      a: remoteAddress. {
        Socket s = #produce(Socket, @open);
    }
}
```

```
public class Client {
    void m(SocketAddress a)
      a: remoteAddress. {
        Socket s = new Socket();
        s.bind(a);
        s.connect();
    }
}
```

(The specifics of code generation will be discussed later)

Transform-queries

```
public class Socket {
    resource state {
       properties @raw, @bound, @open, @closed;

       Socket() this: ++@raw. { }
       void bind(SocketAddress bindPoint) this: -@raw, ++@bound;
         bindPoint:remoteAddress. { }
       void connect() this: -@bound, ++@open. { }
       void send(byte[] data) this: @open; data: ++sentData. { }
    }
}
```

Generate and substitute

```
public class Client {
    void m(Socket s) s: @open. {
       byte[] d = new byte[10000];
       setData(d);
       #transform(d, sentData);
    }
}
```

```
public class Client {
    void m(Socket s) s: @open. {
       byte[] d = new byte[10000];
       setData(d);
       s.send(d);
    }
}
```


Monday, 16 January 2012

Good for side effects

Label signatures

```
public class Socket {
    resource state {
       properties @raw, @bound, @open, @closed;

       Socket() this: ++@raw. { }
       void bind(SocketAddress bindPoint) this: -@raw, ++@bound;
         bindPoint:remoteAddress. { }
       void connect() this: -@bound, ++@open. { }
       void send(byte[] data) this: @open; data: ++sentData. { }
    }
}
```

```
public class SocketUser {
    void m(Socket s) s: -@raw, +@bound, +@open. {
       s.bind(getAddress());
       s.connect();
    }
}
```

++@x: directly added property

+@x: indirectly added property (checkable!)

The method m is now described in terms of its aggregate effects.
It can now also be used to satisfy a query.

Lower bound gives flexibility

```
public class Socket {
    resource state {
      properties @raw, @bound, @open, @closed, @fast;

      Socket() this: ++@raw. { }
      void bind(SocketAddress bindPoint) this: -@raw, ++@bound;
        bindPoint:remoteAddress. { }
      void connect() this: -@bound, ++@open, +@fast. { }
      void send(byte[] data) this: @open; data: ++sentData. { }
    }
}
```

```
public class SocketUser { //@fast is missing
    void m(Socket s) s: -@raw, +@bound, +@open. {
       s.bind(getAddress());
       s.connect();
    }
}
```

The *m* method contract does not need to report all established labels, as long as preconditions (-@x) and invariants (@x) are fully reported

The method m is now described in terms of its aggregate effects.
It can now also be used to satisfy a query.

Resources

```
public class Socket {
    resource state {
       properties @raw, @bound, @open,
@closed;

       String remoteHost = null;
       boolean isConnected = false;
       int connectionSpeed = 0;
      //...
    }

    resource speed {
      properties @fast, @slow;
      int dataSpeed;

      void setFast() this: ++@fast. {
        dataSpeed = 100;
      }
      void setSlow() this: ++@slow. {
        dataSpeed = 10;
      }
    }
}
```

- Directly inspired by *abstract regions* in Boyland-Greenhouse system - use to **avoid unwanted interference**
- Group related data and properties
- Properties may be a predicate on the internal data in the resource => *internal semantic contract*
- When the data in the resource is changed, we say that the resource is *mutated*

Resource mutations must be declared

```
public class Socket {
    resource state {
       properties @raw, @bound, @open, @closed;
       String remoteHost = null;
       boolean isConnected = false;
       //...
    }

    resource speed {
      properties @fast, @slow;
      int dataSpeed;
      void setFast() this: ++@fast. {
        dataSpeed = 100;
      }
      void setSlow() this: ++@slow. {
        dataSpeed = 10;
      }
    }

    void disconnectAndStop() mutates this.speed, this.state:
      this: ++@halted. {
      this.dataSpeed = 0; //Poplar will force these writes to be reported
      this.isConnected = false;
      this.remoteHost = null;
    }
}
```


Note that direct writes to these fields are only permitted in the current formalisation if we also have a ++@x property for that resource.
Note implicit mutations here.

Mutation summary

- Interpretation of a resource mutation: **all properties in that resource are lost,** except for those specified in the label signature
- A set of resource mutations is called a mutation summary. This is:
  - An **upper bound on lost labels**
  - Compositional in the same way as label signatures

Method contract = label signature (lower bound) + mutation summary (upper bound)

Putting it together

```
public class Socket {
    resource state {
       properties @raw, @bound, @open, @closed;

       Socket() this: ++@raw. { }
       void bind(SocketAddress bindPoint) this: -@raw, ++@bound. { }
       void connect() this: -@bound, ++@open. { }
       void send(byte[] data) this: @open; data: ++sentData. { }
    }
    resource speed {
      properties @fast, @slow;

      void setFast() this: ++@fast. { }
      void setSlow() this: ++@slow. { }
    }
}

class SocketUser {
    void m(Socket s) mutates s.state, s.speed:
      s: -@raw, +@open, +@fast. {
      s.bind(getAddress());
      s.connect();
      s.setFast();
    }

}
```


Note that if we add more properties to the Socket, generally, a mutation summary remains true

Uniqueness

- Aliasing is an essential difficulty with languages that have pointers
  - Given two pointers, do they point to the same objects?
- Simple approach: uniqueness kinds - classify references according to assumptions and guarantees[1,2]

1. Minsky, N. *Towards Alias-Free Pointers*. ECOOP 1996
2. Boyland, J. *Alias Burying: Unique Variables Without Destructive Reads*. Softw. Pract & Exp., 2000

Uniqueness kinds

Kind	**Assumption**	**Guarantee**
Normal	None (may be aliased)	None
Unique	Is unique	Remains unique
Maintain	None (may be aliased)	Remains unique

Note that this is not novel – these ideas are well known in the literature, but under slightly different names

Uniqueness and mutations

```
class SocketUser {
    void m(Socket s) mutates s.state, s.speed:
      s: maintain, -@raw, +@open, +@fast. {
      s.bind(getAddress());
      s.connect();
      s.setFast();
    }

    void withUnique(Socket u) mutates u.state, u.speed:
      u: unique. {
      m(s);
    }
    void withAliases(Socket a) mutates any(Socket).state,
      any(Socket).speed: { //a is implicitly a “normal” variable
      m(a);
    }
    void withNew() { //No need to report anything
      m(new Socket());
    }
}
```

The reported mutations are different depending on the uniqueness kinds of the variables passed to a method.

This is one of the major sources of imprecision.

Design - summary

- Labels as a least unit of specification
- Resources group properties and related state
- Label signatures give a lower bound on established state
- Mutation summaries give an upper bound on erased labels
- Uniqueness kinds to handle aliasing

Design - justification

- **Sufficient** features to describe Java code as an AI planning domain for practical purposes (to be demonstrated)
- **Necessary** features
  - Temporal constraints (properties) must be addressed
  - Interference (resources) must be addressed
  - Queries needed to request an integration
  - Aliasing (uniqueness kinds) must be addressed

When I had to make design choices in order to merge the various elements, I tried firstly to make it simple, second, to make use of opportunities to make components more evolvable.

Comparison

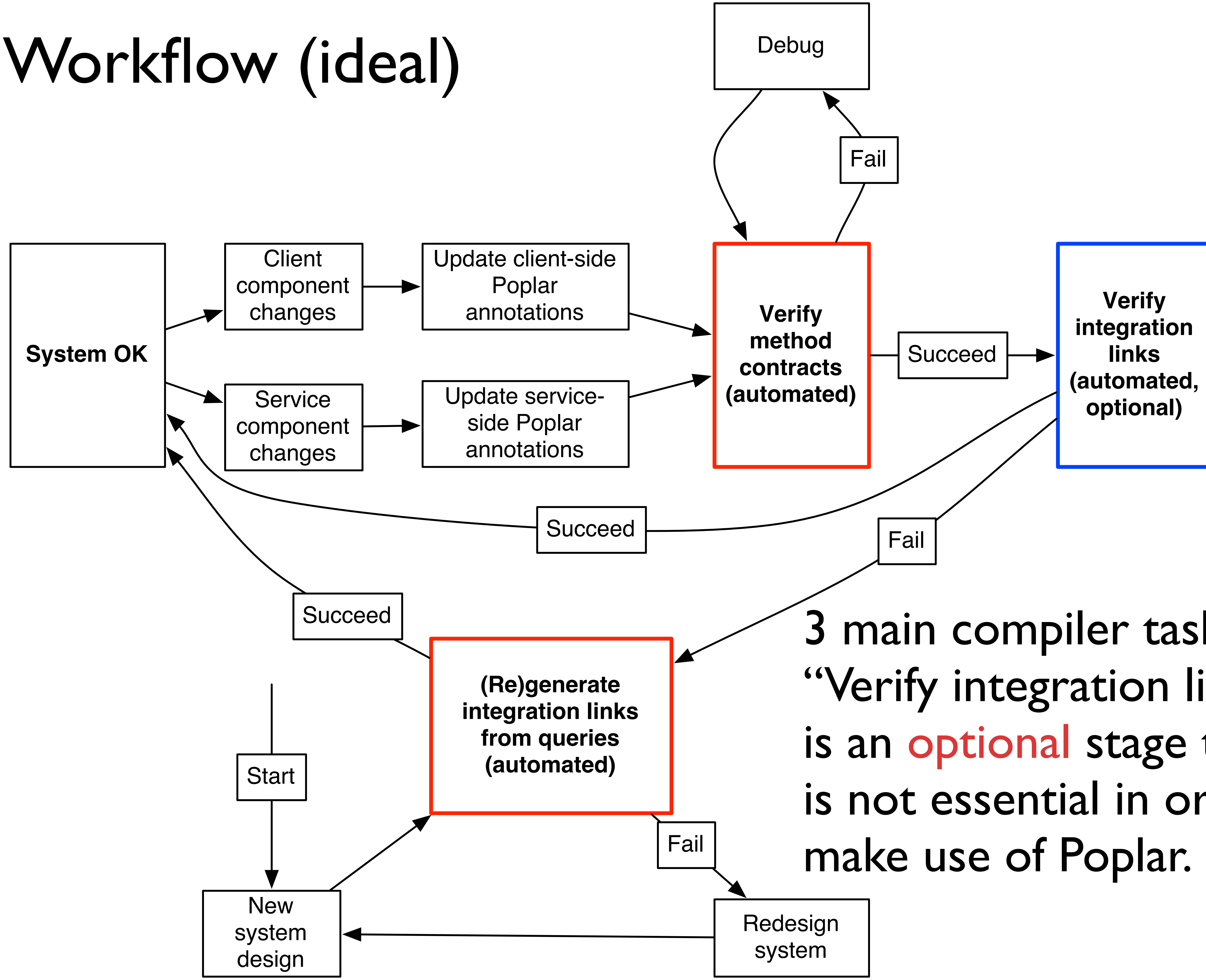
System	Poplar	B/G Effect	Typestate/Fugue	Labelled LC	Prospector
Polymorphic regions	✓	✓			
Subregions		✓			
Effect summaries	✓	✓			
Temporal state names	✓		✓		
Labelled arg. selection	✓			✓	
State for individual frames			✓		
Type-based queries	✓				✓
Search/AI planning	✓				✓
Unique pointers	✓	✓	✓		
Static checking	✓	✓	✓		

Poplar is not a complete replacement for any of the systems we have borrowed from, rather it is a compromise between different designs

Another perspective

Poplar works by breaking down the contract of each method into small units, and reasoning about these individually

Workflow (ideal)



3 main compiler tasks.
"Verify integration links"
is an optional stage that
is not essential in order to
make use of Poplar.

Monday, 16 January 2012

Could add some colour here, etc
describe relative importance of stages

Outline

- Introduction
- Design
- Demo
- Formalisation
- Implementation
- Case study and evaluation
- Conclusion

25 min mark

Outline

- Introduction
- Design
- Demo
- Formalisation
- Implementation
- Case study and evaluation
- Conclusion

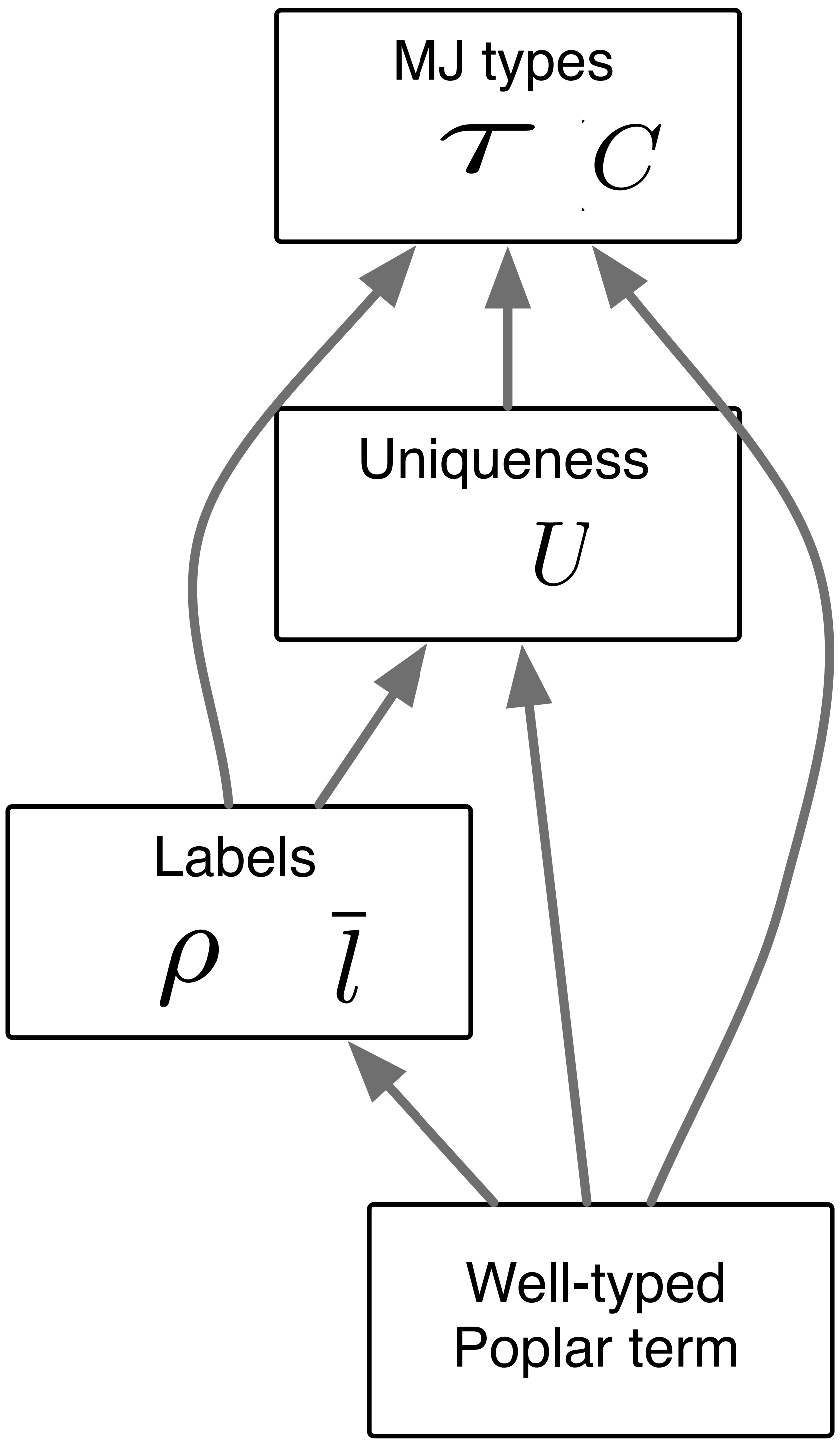
29 min mark

MJ

- Core calculus for an imperative fragment of Java[1, 2]
- Models mutable state and Java's block structure faithfully
- Valid subset of Java
- Boyland/Greenhouse effect system has already been studied in the context of MJ

1. Bierman, G.M., Parkinson, M.J., and Pitts, A.M. *MJ: An Imperative Core Calculus for Java and Java with Effects*. Tech report Cambridge U., 2003
2. Bierman, G.M. and Parkinson, M.J. *Effects and Effect Inference for a Core Java Calculus*. WOOD 2003.

Big picture



- Formalisation based on MJ
- Poplar types = MJ (Java) types + uniqueness kinds + labels and effects
- Well-typed Poplar terms are guaranteed to use labels correctly (to be defined)

Formalisation structure

- Judgments for
  - Well-formed class
  - Well-formed overriding
  - Poplar typing for statements, expressions
    - Labels, mutations, uniqueness
  - Valid solution to a query

Composing method contracts (chaining)

```
void configure(Address a) mutates this.configuration:
  this: +@configured, @notConnected. { ... }
```

+

```
void connect(Address a) mutates this.connection:
  this: +@connected, @configured;
```

=

```
configure(a);
connect(a);
```

```
mutates this.connection, this.configuration:
  this: +@connected, +@configured;
```

When statements are executed in sequence, we can obtain a contract for the resulting fragment

Soundness

- ***Establishment* of a label**: being created by a method annotated with ++t or ++@p
- **Use of a label:** being assumed as a precondition for some statement
- **A Poplar fragment is sound if all labels for all values are**
  - Established before they are used
  - Not erased between the point of establishment and the point of use

Soundness (2)

- I believe that the Poplar type system is sound - a proof is left for future work
  - One possibility is altering the semantics to model creation and destruction of labels directly

Technical achievements

- **Polymorphism of properties** (subclasses can redefine or extend meaning)
- **Polymorphism of resources** (subclasses can redefine, add new properties)
- **Modular** checking and compilation

A limitation

```
class Base {
 resource r {
  properties @p;
  int i;
  void makeP() this: ++@p. {
    i = 0;
  } }
}

class E1 extends Base {
 resource r {
   int j;
   //Stronger invariant for @p
   void makeP() this: ++@p. {
     super.makeP();
     useP(); //Invalid!
     j = 0;
   }

   void useP() this: @p. {
   } }
}
```

- Properties that are overloaded by subclasses are handled in a restricted way
  - Must be established in *all* class frames before they can be used
  - Some typestate systems[1] track states in each frame independently

1. Deline, R. and Fähndrich, M. *Typestates for Objects*. ECOOP 2004.

(Very small) example

$$\Delta;\Gamma \vdash s : \tau + \mathsf{LS}!\rho$$ Statement type

$(\mathsf{LS} \stackrel{\text{def}}{=} (\mathsf{LS}_+, \mathsf{LS}_=, \mathsf{LS}_-))$ Label signature

$$\Delta;\Gamma \vdash e!\emptyset \text{ ok} \qquad \Delta;\Gamma \vdash x!\overline{r} \text{ ok} \qquad \Delta;\Gamma \vdash \mathsf{this}.f!\overline{r} \text{ ok} \qquad \frac{\Delta;\Gamma \vdash e!\overline{r} \text{ ok}}{\Delta;\Gamma \vdash (C)e!\overline{r} \text{ ok}}$$

$$\frac{\Delta;\Gamma \vdash e : C : \mathsf{Fresh}}{\Delta;\Gamma \vdash e!\overline{r} \text{ ok}}$$ Acceptable mutation

$$U \rightsquigarrow U \qquad \mathsf{Normal} \rightsquigarrow \mathsf{Maintain} \qquad \mathsf{Unique} \rightsquigarrow \mathsf{Maintain} \qquad \mathsf{Fresh} \rightsquigarrow U$$

Uniqueness

TS-SEQVARWRITE

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash x = e; : \mathsf{void} + \mathsf{LS}_1!\rho_1 \\ \Delta;\Gamma \vdash s_2 \ldots s_n : \tau + \mathsf{LS}_2!\rho_2 \\ \Delta;\Gamma \vdash e!\rho_2(x) \text{ ok} \qquad \Delta;\Gamma \vdash e : C : U \\ (\mathsf{LS}_1, \rho_1) \oplus (lflow(\mathsf{LS}_2, x, e), rflow(\rho_2, x, e, C, U)) = (\mathsf{LS}, \rho)\end{array}}{\Delta;\Gamma \vdash x = e; s_2 \ldots s_n : \tau + \mathsf{LS}!\rho}$$

Statement type for var. write sequences

Formalisation summary

- Based on MJ
- Extended type system describes and constrains the Poplar concepts
  - A well-typed Poplar fragment is, when compiled, a well-typed MJ fragment
- Soundness proof not yet done

Outline

- Introduction
- Design
- Demo
- Formalisation
- Implementation
- Case study and evaluation
- Conclusion

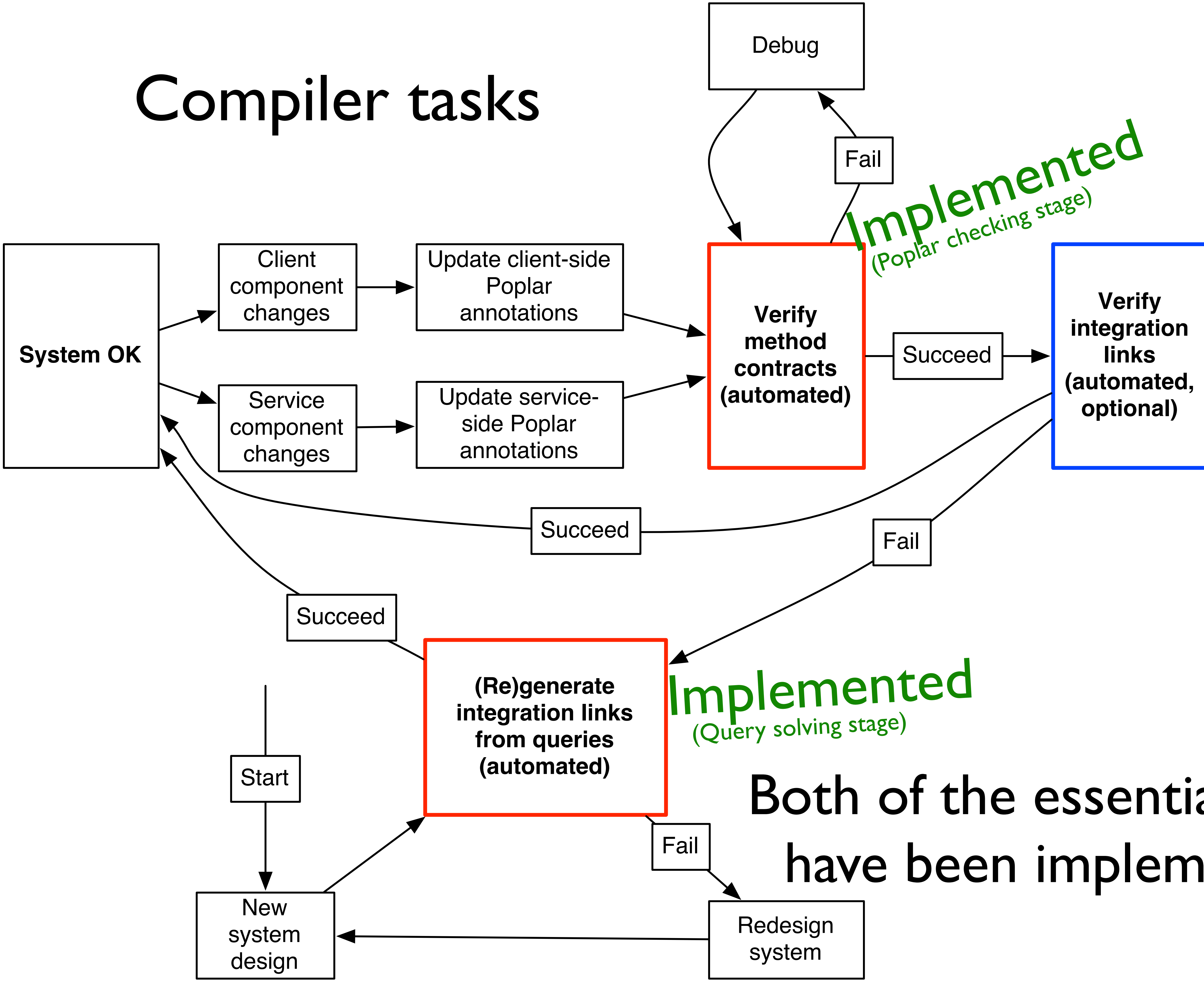
34 min mark

Jardine

- A Poplar compiler, Jardine, has been implemented by extending JKit, a Java compiler[1]
    - Alexandre Pichot contributed to the grammar, parser and uniqueness system (mainly), the rest implemented by me
- Poplar checking and generation of integration links are implemented, except:
    - Some remaining work in uniqueness handling
    - Valid overriding is not checked

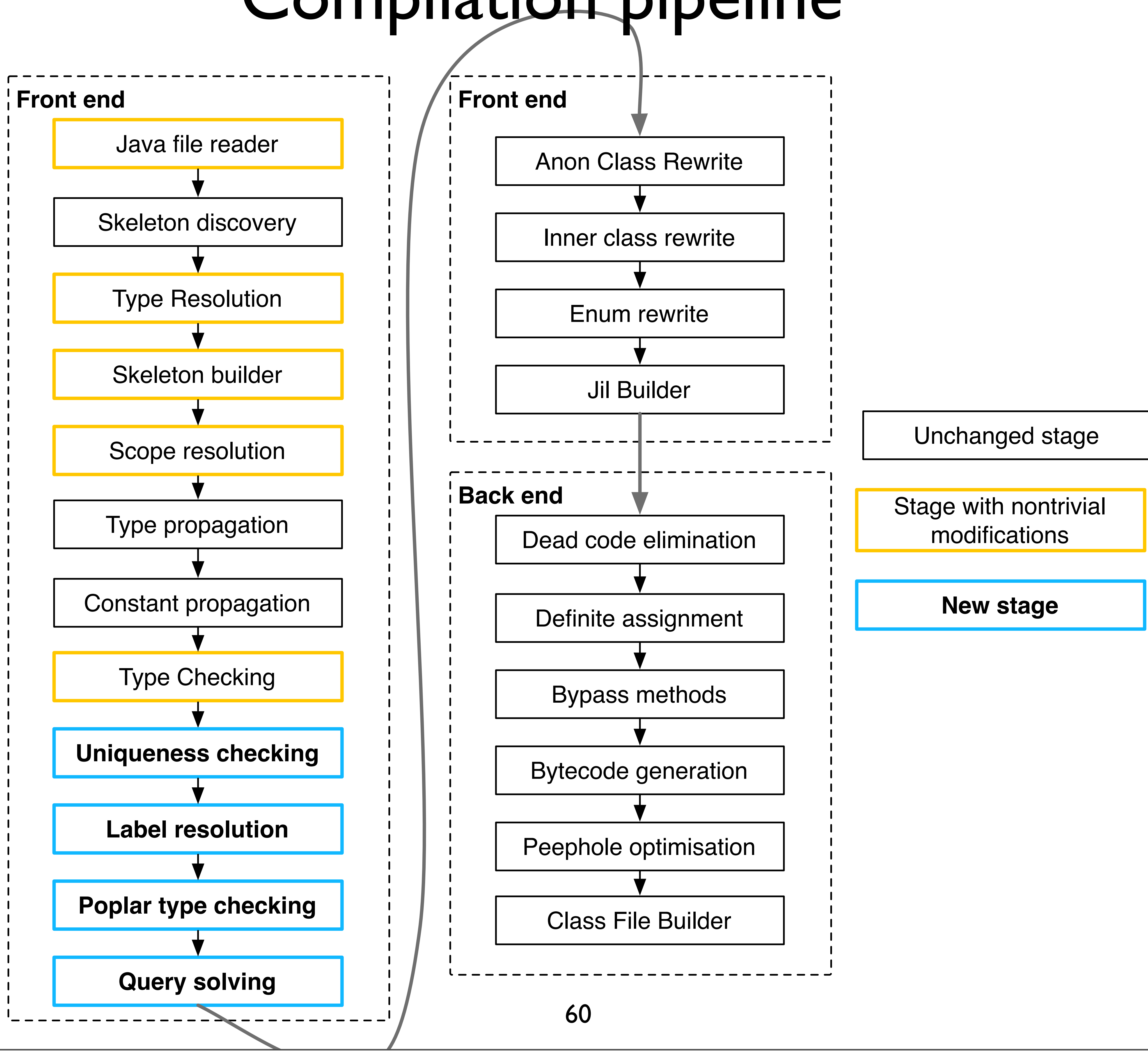
1. Pearce, David J. *JKit*. http://homepages.ecs.vuw.ac.nz/~djp/jkit. 2011.

Compiler tasks



Both of the essential stages have been implemented.

Compilation pipeline



Poplar checking stage

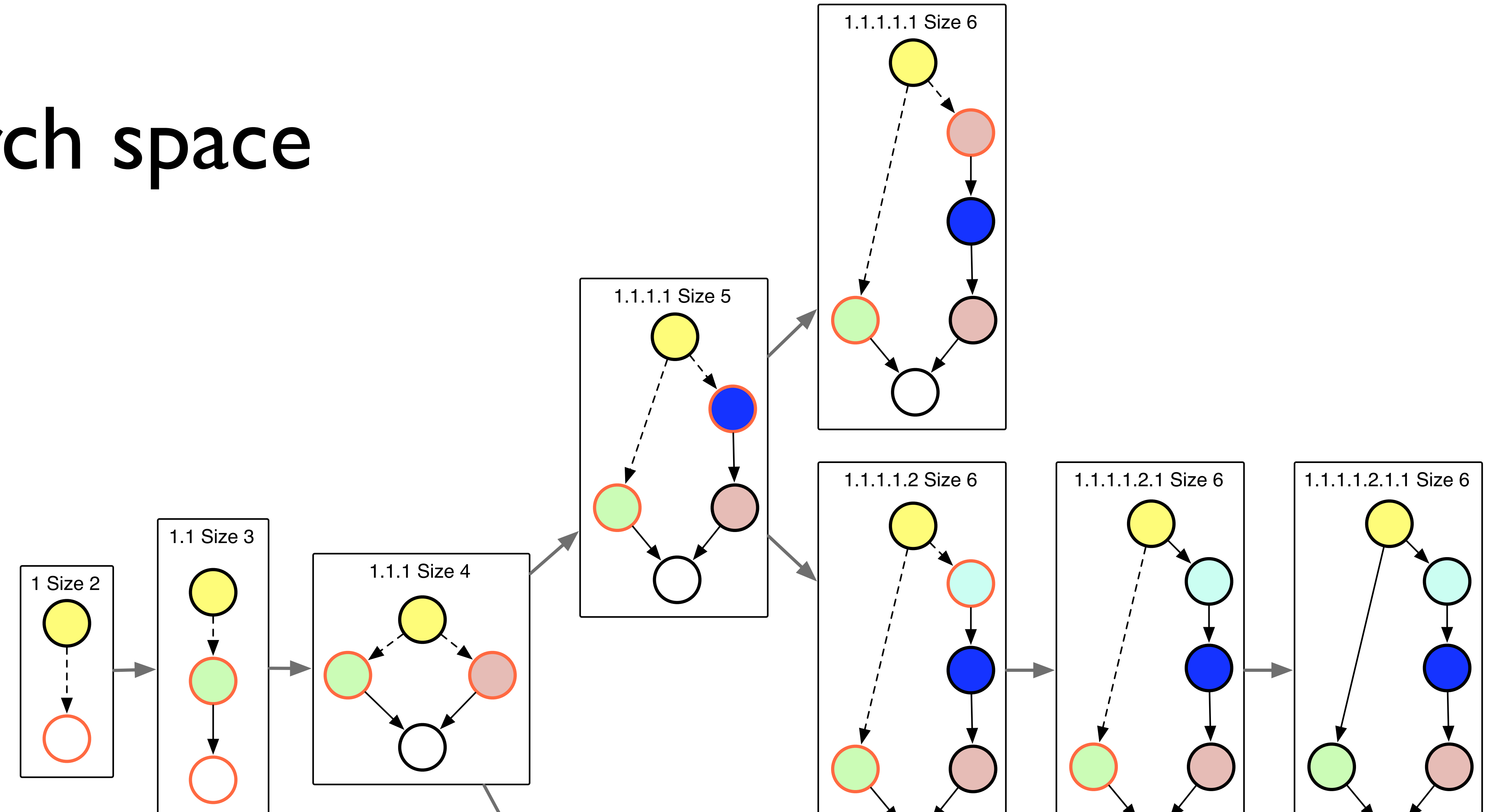
- Implements the formalised Poplar type system
- Reconstructs the type of every term and statement, verifying that there is some way to satisfy all label requirements

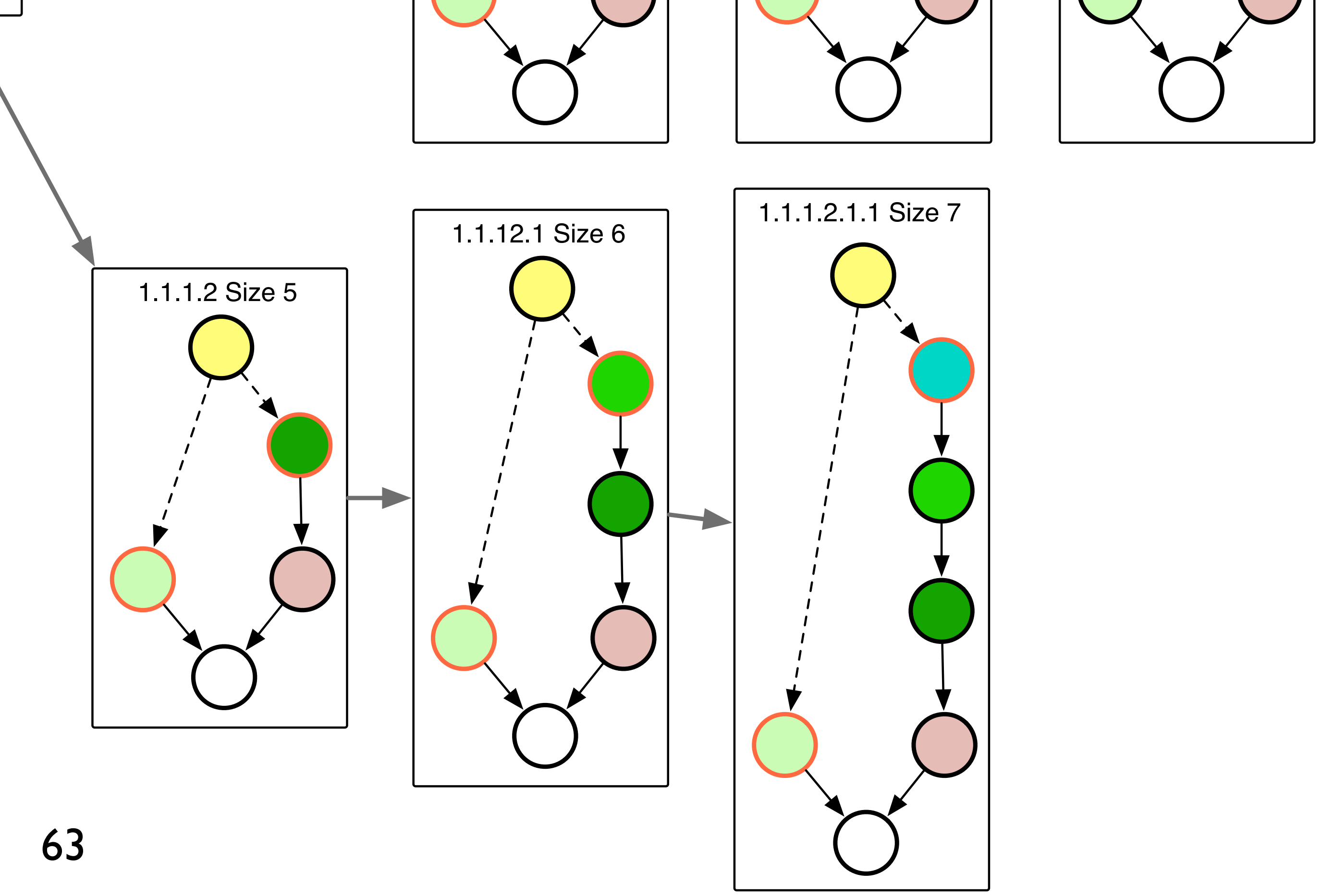
Query solving stage

- Uses **Partial Order Planning** (POP)[1,2] to find solutions to queries - but in theory, any planning algorithm may be used
- Replaces queries by their solutions
- We search the space of **well typed Poplar fragments**

1. McAllester, D. and Rosenblitt, D. *Systematic Nonlinear Planning*. Nat. Conf. on AI, 1991
2. Nguyen, X. and Kambhampati, S. *Reviving Partial Order Planning*. 17 Intl. Joint Conf on AI, 2001

Search space





- Always prefer small solutions over large ones
- A progress measure guarantees that we do not get stuck in an infinite loop
  - There is only ever a finite amount of progress that can possibly be constructed

Progress measure

- Expressed in terms of **open preconditions**
- We make progress if we create a new precondition set that is not a superset of a previously achieved set
- Open preconditions are expressed in terms of **labels and types**, but should eventually also track uniqueness

Outline

- Introduction
- Design
- Demo
- Formalisation
- Implementation
- Case study and evaluation
- Conclusion

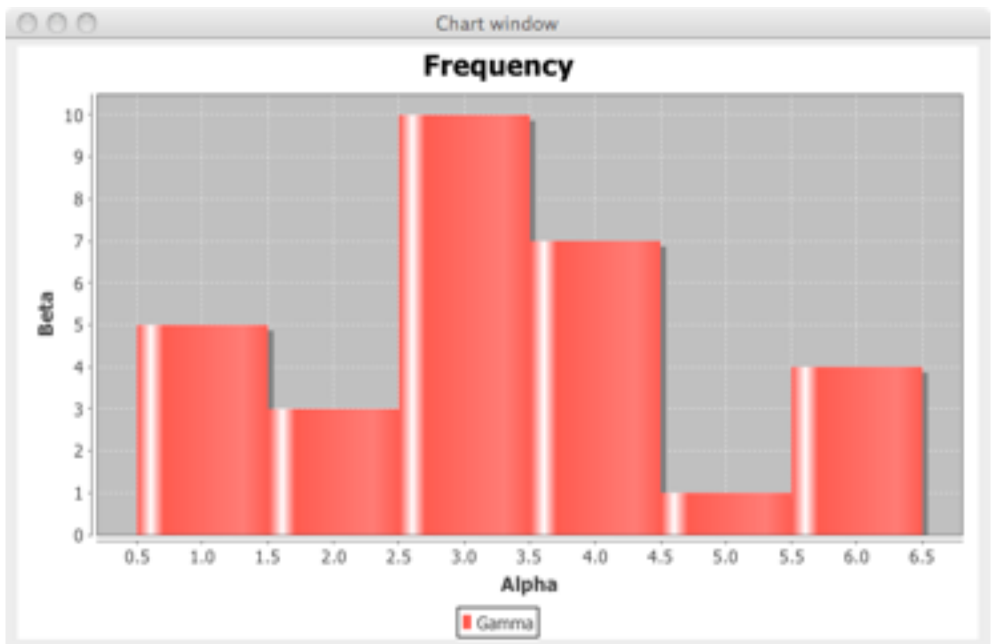
39 min mark

Case study

- We will study JFreeChart[1], a popular Java chart library
- Goal: demonstrate that we can use Poplar with an existing codebase
- We will gain the freedom to refactor JFreeChart dramatically without disturbing API clients

1. Gilbert, D. et. al. *JFreeChart.* http://www.jfree.org/jfreechart. 2011.

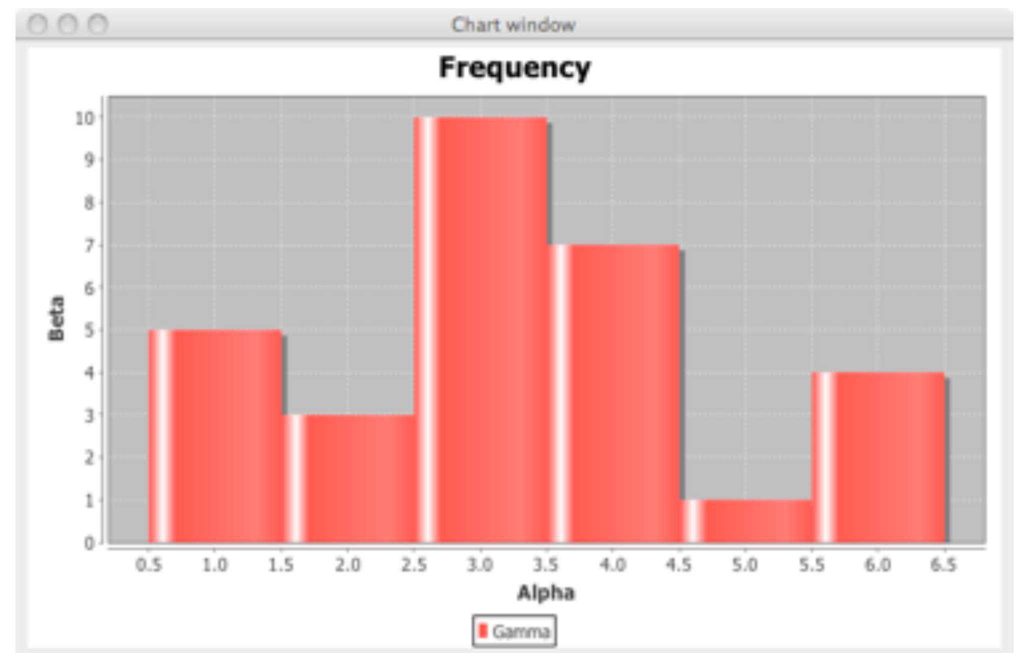
Using JFreeChart




```java
class ChartClient {
  /* makeChartFrame(), makeChart(points) have been omitted */

  private static JComponent makeChart(Collection<Integer> points) {
    XYSeriesCollection dataSet = new XYSeriesCollection();
    XYSeries s1 = new XYSeries("Gamma");

    int x = 0;
    for (Integer i : points)
    {
      x++;
      s1.add(x, i);
    }

    dataSet.addSeries(s1);

    JFreeChart chart = ChartFactory.createXYBarChart("Frequency",
      "Alpha", false, "Beta", dataSet,
      PlotOrientation.VERTICAL, true, true, false);

    return new ChartPanel(chart);
  }

  public static void main(String[] args)
  {
    Collection<Integer> points = getData(args);
    JFrame frame = makeChartFrame();
    JComponent chart = makeChart(points);
    JPanel c = new JPanel();
    c.add(chart);
    frame.setContentPane(c);
    frame.setVisible(true);
  }
}
```

Using JFreeChart

```
private static JComponent makeChart(Collection<Integer> points) {
  XYSeriesCollection dataSet = new XYSeriesCollection();
  XYSeries s1 = new XYSeries("Gamma");

  int x = 0;
  for (Integer i : points)
  {
    x++;
    s1.add(x, i);
  }

  dataSet.addSeries(s1);

  JFreeChart chart = ChartFactory.createXYBarChart("Frequency",
    "Alpha", false, "Beta", dataSet,
    PlotOrientation.VERTICAL, true, true, false);

  return new ChartPanel(chart);
}
```



Integrating with a query

Client

```
private static JFreeChart useFactoryIndirect(XYSeriesCollection
    dataSet)
  dataSet: tGenChartData. {
  String title:(tChartTitle) = "Frequency";
  PlotOrientation po:(tPlotOrientation) = PlotOrientation.VERTICAL;
  String f:(tXAxisLabel, tCategoryAxisLabel) = "Alpha";
  String a:(tYAxisLabel, tValueAxisLabel) = "Beta";
  boolean da:(tWithDateAxis) = false;
  boolean gu:(tGenUrls) = false;
  boolean tt:(tGenTooltips) = true;
  boolean lr:(tReqLegend) = true;

  /* Produce the chart using a query */
  JFreeChart c = #produce(JFreeChart, tXYBarChart);
  return c;
}
```

Library

```
public static JFreeChart createXYBarChart(String title,
  String xAxisLabel, boolean dateAxis,
  String yAxisLabel, IntervalXYDataset dataset,
  PlotOrientation orientation, boolean legend,
  boolean tooltips, boolean urls)
    title: tChartTitle; dateAxis: tWithDateAxis;
      xAxisLabel: tXAxisLabel; yAxisLabel: tYAxisLabel;
      orientation: tPlotOrientation; legend: tReqLegend;
      tooltips: tGenTooltips; dataset: tGenChartData;
      urls: tGenUrls;
      result: ++tXYBarChart.
    { ... }
```

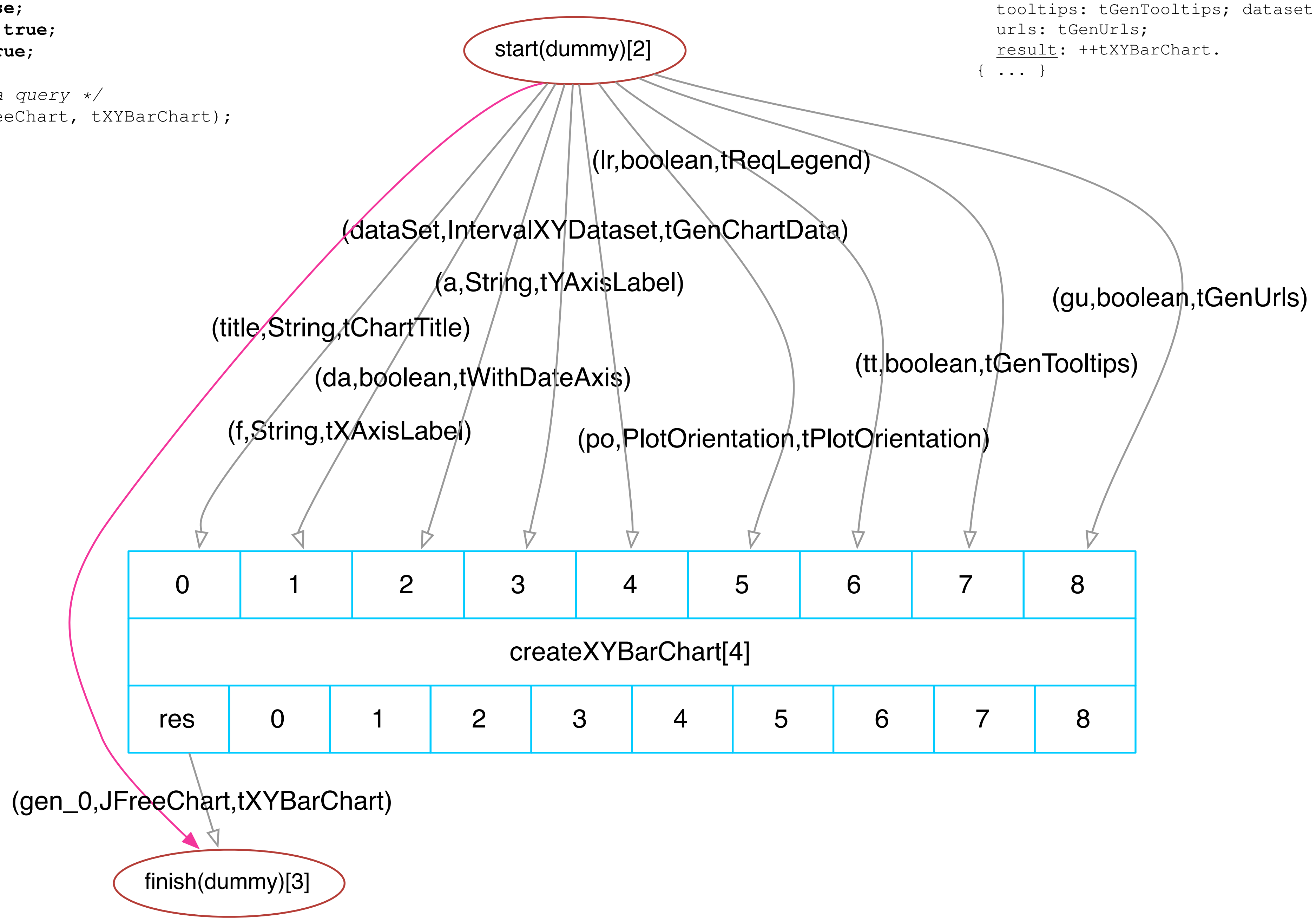
Result

Client

```
private static JFreeChart useFactoryIndirect(XYSeriesCollection
    dataSet)
  dataSet: tGenChartData. {
  String title:(tChartTitle) = "Frequency";
  PlotOrientation po:(tPlotOrientation) = PlotOrientation.VERTICAL;
  String f:(tXAxisLabel, tCategoryAxisLabel) = "Alpha";
  String a:(tYAxisLabel, tValueAxisLabel) = "Beta";
  boolean da:(tWithDateAxis) = false;
  boolean gu:(tGenUrls) = false;
  boolean tt:(tGenTooltips) = true;
  boolean lr:(tReqLegend) = true;

  /* Produce the chart using a query */
  JFreeChart c = #produce(JFreeChart, tXYBarChart);
  return c;
}
```

Library

```
public static JFreeChart createXYBarChart(String title,
  String xAxisLabel, boolean dateAxis,
  String yAxisLabel, IntervalXYDataset dataset,
  PlotOrientation orientation, boolean legend,
  boolean tooltips, boolean urls)
    title: tChartTitle; dateAxis: tWithDateAxis;
      xAxisLabel: tXAxisLabel; yAxisLabel: tYAxisLabel;
      orientation: tPlotOrientation; legend: tReqLegend;
      tooltips: tGenTooltips; dataset: tGenChartData;
      urls: tGenUrls;
      result: ++tXYBarChart.
    { ... }
```




```
JFreeChart gen_0 = ChartFactory.createXYBarChart(title, f, da, a,
    dataSet, po, lr, tt, gu);
```

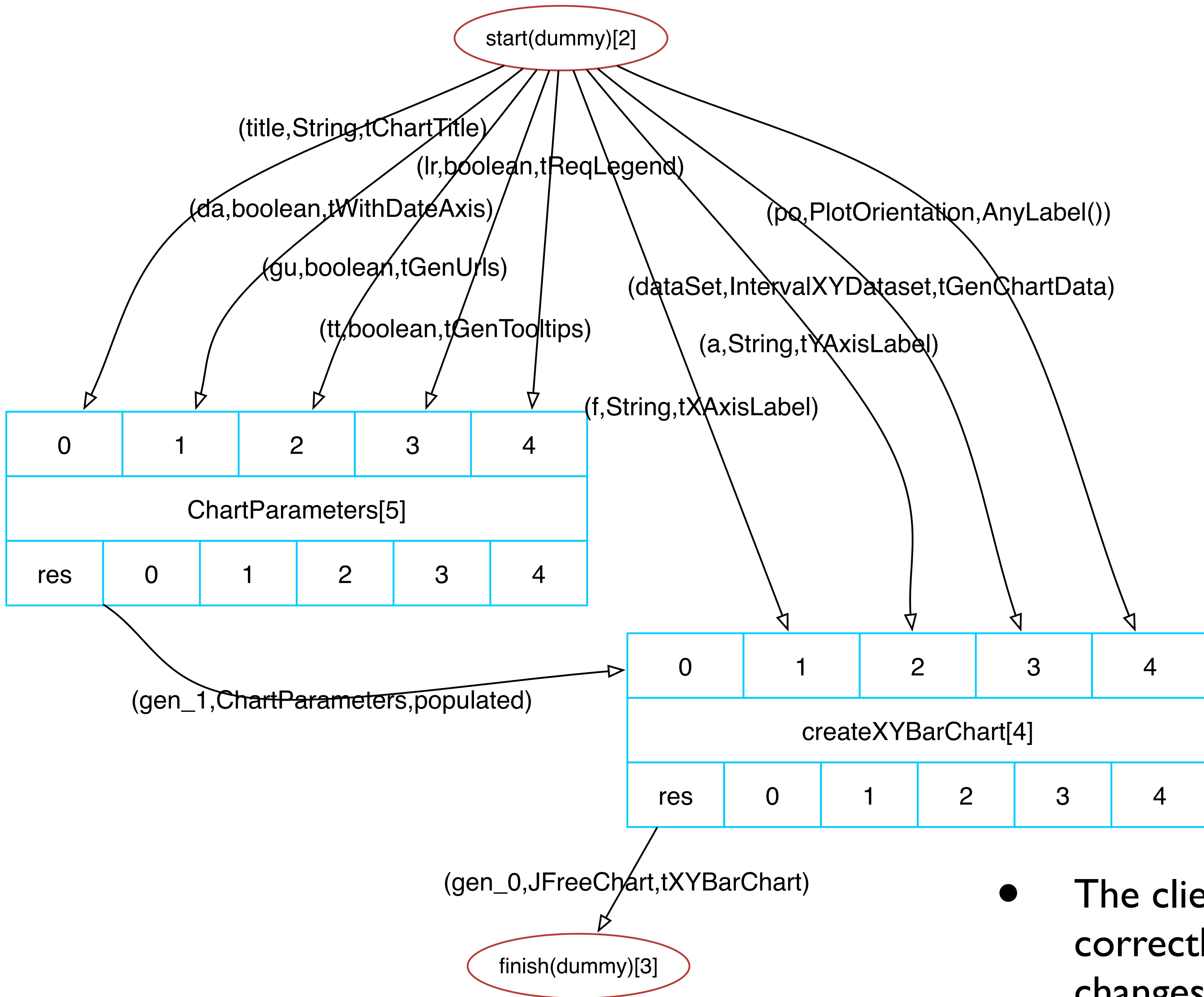
A parameter object

```
public class ChartParameters {

  tag(ChartParameters) populated;

  String chartTitle;
  boolean withDateAxis, urls, tooltips, legend;

  public ChartParameters(String chartTitle, boolean withDateAxis,
      boolean urls, boolean tooltips, boolean legend)
    result: ++populated; chartTitle: tChartTitle;
    withDateAxis: tWithDateAxis; urls: tGenUrls;
    tooltips: tGenTooltips; legend: tReqLegend. {
    this.chartTitle = chartTitle; this.withDateAxis = withDateAxis;
    this.urls = urls; this.tooltips = tooltips;
    this.legend = legend;

  }
}

public class ChartFactory {
  //etc.

  public static JFreeChart createXYBarChart(ChartParameters cp,
  String xAxisLabel, String yAxisLabel,
  IntervalXYDataset dataset, PlotOrientation orientation)

      cp: populated;
      xAxisLabel: tXAxisLabel;
      yAxisLabel: tYAxisLabel;
      dataset: tGenChartData;
      result: ++tXYBarChart. { ... }

//etc.
}
```

- Change library API
- Instead of passing several parameters individually, pass them in a containing object
- This refactoring is recommended by Fowler[1] for certain situations

1. Fowler, M. *Refactoring: Improving the Design of Existing Code*. 1999.

Result



- The client is updated correctly without any manual changes

```
ChartParameters gen_1 = new ChartParameters(title, da, gu, tt, lr);
JFreeChart gen_0 = ChartFactory.createXYBarChart(gen_1, f, a, dataSet,
    po);
```

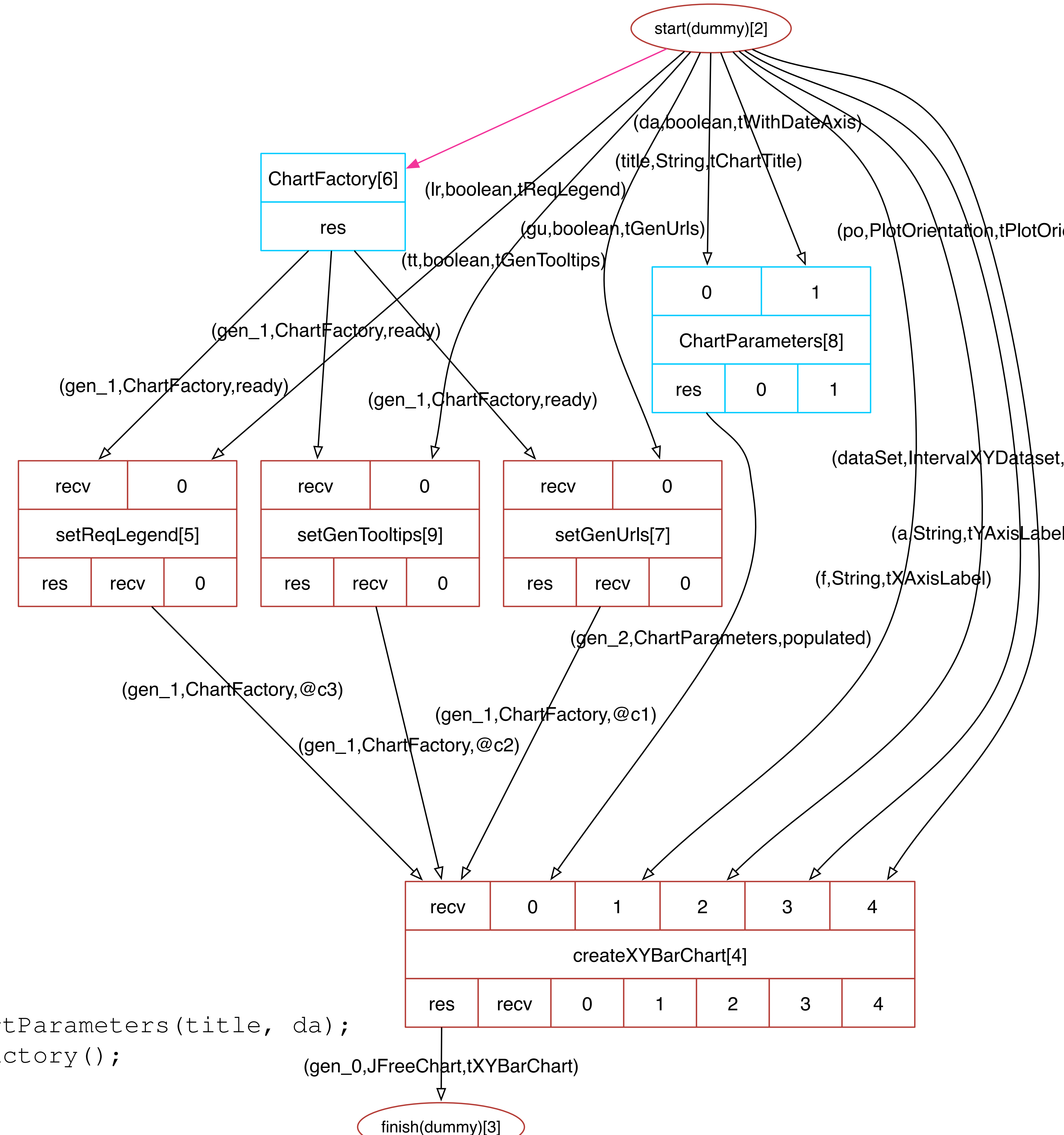
Converting parameters to state

```
class ChartFactory {
//etc.
  composite @factoryConfigured = (@c1, @c2, @c3);

  public ChartFactory() result: ++ready. { }

  resource urlConfig {
    properties @c1;
    private static boolean currentUrls;
  }
  resource legendConfig {
    properties @c2;
    private static boolean currentLegend;
  }
  resource tooltipsConfig {
    properties @c3;
    private static boolean currentTooltips;
  }

  public void setGenUrls(boolean urls)
    urls: tGenUrls;
    this: ready, ++@c1. {
    currentUrls = urls;
  }
  public void setGenTooltips(boolean tooltips)
    tooltips: tGenTooltips;
    this: ready, ++@c2. {
    currentTooltips = tooltips;
  }
  public void setReqLegend(boolean legend)
    legend: tReqLegend;
    this: ready, ++@c3. {
    currentLegend = legend;
  }
  public  void resetConfiguration() mutates urlConfig, legendConfig,
      tooltipsConfig: {
    currentLegend = false;
    currentTooltips = false;
    currentUrls = false;
  }

  public JFreeChart createXYBarChart(ChartParameters cp,
  String xAxisLabel, String yAxisLabel, IntervalXYDataset dataset,
  PlotOrientation orientation)
        cp: populated; xAxisLabel: tXAxisLabel;
        yAxisLabel: tYAxisLabel; dataset: tGenChartData;
        orientation: tPlotOrientation; this: @factoryConfigured;
        result: ++tXYBarChart.
    { ... }
```

- Instead of passing parameters, we assign default values (template data) to the factory class
- We require these to be initialised before the factory may be used.

Result

- The client is updated correctly without any manual changes



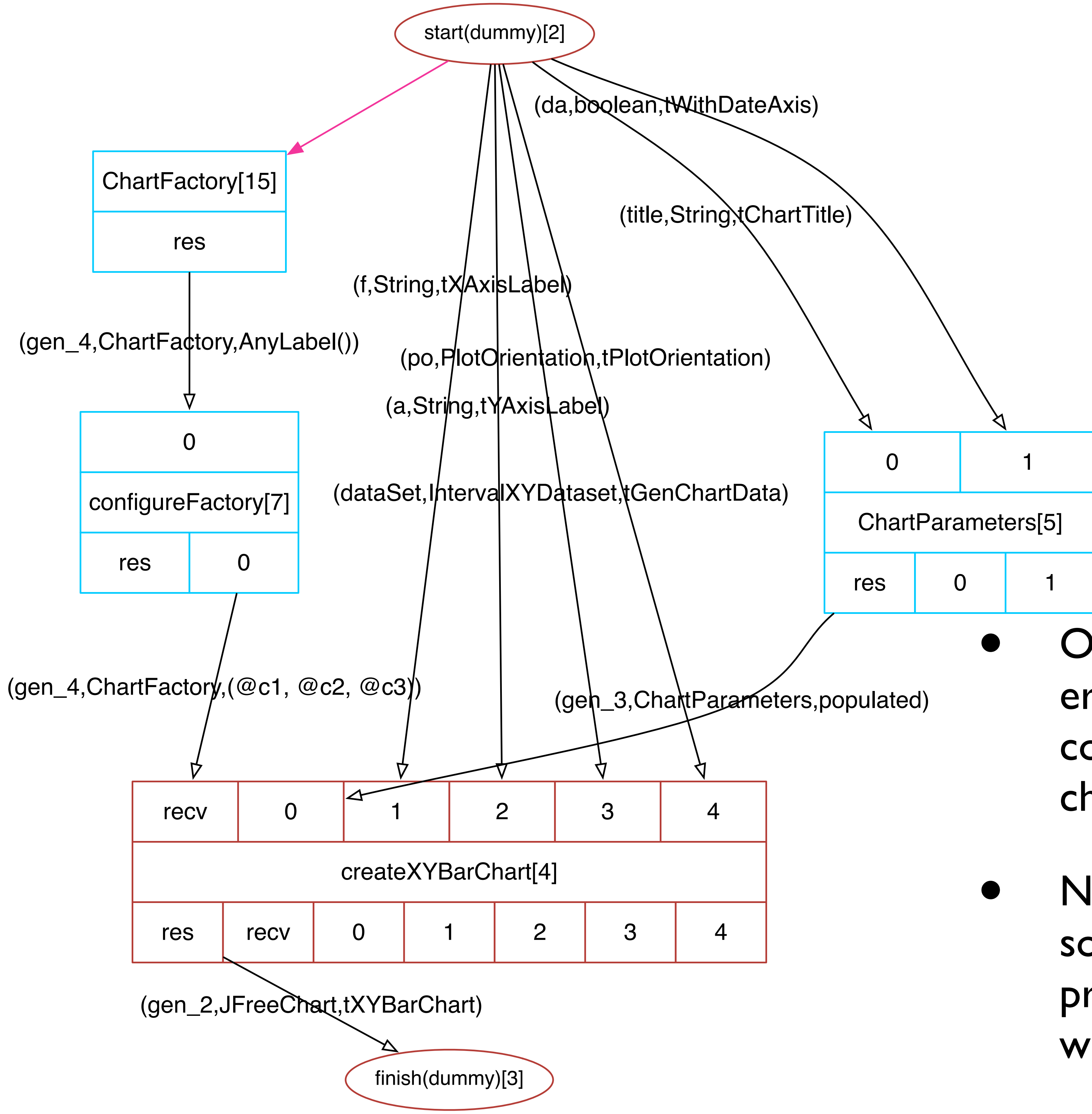

```
ChartParameters gen_2 = new ChartParameters(title, da);
ChartFactory gen_1 = new ChartFactory();
gen_1.setReqLegend(lr);
gen_1.setGenUrls(gu);
gen_1.setGenTooltips(tt);
JFreeChart gen_0 = gen_1.createXYBarChart(gen_2, f, a, dataSet, po);
```

Configuring the factory manually

```
1   class ChartClient {
2   //...
3     private static void configureFactory(ChartFactory cf)
4     cf: +@factoryConfigured. {
5       cf.setGenTooltips(true);
6       cf.setGenUrls(false);
7       cf.setReqLegend(true);
8       cf.resetConfiguration();    //This line violates the method's
            contract
9     }
10  //...
11  }
```

- Instead of relying on Poplar to find configuration parameters, we supply them manually in the client
- This method takes precedence because it results in a shorter solution
- We introduce an error on line 8 (for demo purposes) that will be detected by Poplar

Result



- Once we have removed the error, the client is updated correctly without any manual changes
- Note that this shorter solution is preferred over the previous (longer) solution, which is still valid

```
ChartFactory gen_4 = new ChartFactory();
ChartClient.configureFactory(gen_4);
ChartParameters gen_3 = new ChartParameters();
JFreeChart gen_2 = gen_4.createXYBarChart(gen_3, f, a, po, gen_3);
```

Case study results

- We have demonstrated that Poplar can be used with existing Java libraries to permit a wide range of refactorings without disturbing clients, *once the initial cost of introducing queries has been paid.*

Brute force Poplar conversion

- By generating enough unique label names, we can **always convert an ordinary Java method call or field access into a query** with a predictable result
  - However, protecting the established state and designing resources well may not always be possible with a "naive conversion"

Discussion

- Achievements
- Limitations

Remark

- Three roles of a label: **external semantic contract**, temporal contract, internal semantic contract
- For each individual label, the **external semantic contract** must be preserved or strengthened across versions of components

This is very attractive, because now we have gone from several high level and large scale constraints (on the level of methods, fields) to a more fine grained constraint (on the level of semantic units)

Achievements

- The goal has been to allow Java components to evolve while remaining integrated
- Sensitivities
  - Syntactic/structural changes
  - Semantic changes
  - Temporal constraint changes

Achievements: evolvability

- **Structural and temporal changes become almost irrelevant.**
    - As long as we can construct a path from the starting state to the requested goal state, we can compensate for these changes (see JFreeChart study)
- **Semantic changes to methods and fields become irrelevant, if labels are preserved correctly**

“Almost irrelevant” – in the sense that some changes may lead to other solutions being found first, which may not be irrelevant

Evolution consequences

Service/client side	Disturbance to mutation summaries	Manual client changes necessary	Solutions may change	Compilation may be impossible
Add property			✓	
Remove property			✓	✓
Strengthen label contract (ext. semantic)				
Weaken label contract (ext. semantic)		✓		
Move property to different resource	If explicit method calls exist	If explicit method calls exist	✓	✓
Change temporal contracts			✓	✓
Change internal property contracts				
Add mutation to mut. summary	If called explicitly	If called explicitly	✓	✓
Remove mutation from mut. summary			✓	

Achivements: design and implementation

- **Object-oriented principles**: encapsulation and polymorphism of properties, resources
- Ability to **describe and work with real software systems**
- Rigorous specification
- Usable implementation

Limitations

- Imprecision
  - Uniqueness system is too restrictive and imprecise
  - Sometimes the return type from a method is expected to be downcast to a different type (see Prospector[1])

1. Mandelin, D., Xu, L., Kimelman, D., and Bodik, R. *Jungloid Mining: Helping to Navigate the API Jungle*. PLDI 2005.

Limitations (2)

- Impossible to request negative effects or prevent labels from being established
  - We may simulate negative state by creating a special property that erases state when “established”
- Method effects and preconditions must be expressed as *conjunctions of atomic facts*
  - Disjunctions of conjunctions would be very simple to implement

Limitations (3)

- Effort in writing annotations (however, protocol mining is a well studied problem)
- Data flow between Java and Poplar
  - With a more accurate aliasing system, the user might be able to annotate *all* code (no pure Java)
  - With interop, warnings/guarantees/errors should be easy to implement

Also: a possibility is a static dataflow analysis tool to produce information in auxiliary files, which warn the user about some possible violations

More related work (selected)

1. Alfonso, E.J. *Automatic Protocol-conformance Recommendations*. OOPSLA 2011 poster.

2. Batory, D. and Geraci, B.J. *Composition Validity and Subjectivity in GenVoca Generators*. TSE 1997.

3. Gabel, M. and Su, Z. *Symbolic Mining of Temporal Specifications*. ICSE 2008 (and many others)

4. Kiczales, G.J. et al. *Aspect-Oriented Programming*. 1997

5. Ireland, A. and Stark, J. *Combining Proof Plans with Partial Order Planning for Imperative Program Synthesis*. ASE 2006.

6. Zaremski, A.M. and Wing, J. *Specification Matching of Software Components*. TSE 1997.

7. Becker, S. et al. *Towards an Engineering Approach to Component Adaptation*. LNCS 3938, 2006. (and many others)

8. Jaspan, C and Aldrich, J. *Checking Framework Interactions with Relationships*. ECOOP 2009.

Best location in the story?

Poplar publications

- Rejected
  - ECOOP 2011, POPL 2011, ESOP 2012, ...
    - Many reviewers liked the general approach, but it was probably too early
- Accepted
  - Nyström-Persson, J and Honiden, Shinichi. *Poplar: Java Composition with Labels and AI Planning*. Proc. of the Workshop on Free Composition (FREECO) at Onward! 2011.
- Planned
  - New paper about design, formalism (possibly CBSE, SPLASH, TSE)

Outline

- Introduction
- Design
- Demo
- Formalisation
- Implementation
- Case study and evaluation
- Conclusion

53 min mark (!)

Conclusion

- By combining constraints from various well-studied domains, we can express Java code in such a way that AI planning generates meaningful results
- Hypothesis confirmed
  - AI planning, labels, and a typestate-like formalism may be combined to yield an automatic integration system that is robust to evolution

Some future work

- Accuracy improvements: better aliasing system?
- Finish basic implementation (override checking)
- Implement integration link verification?
- Subresources
- Resource links (needed in practice for many examples, e.g. JDBC)
- Quality metrics for solutions?
- Study more libraries, write applications

Thanks

Yukino Baba, Valentina Baljak, Lodewijk Bergmans, Christoph Bockish, Shigeru Chiba, Daisuke Fukuchi, Levent Gürgen, Masami Hagiya, Ichiro Hasuo, **Shinichi Honiden**, Liyang Hu, Zhenjiang Hu, Atsushi Igarashi, Fuyuki Ishikawa, Ciera Jaspan, Fan Jiang, Gabriel Keeble-Gagnére, Adrian Klein, Benjamin Klöpper, Hidehiko Masuhara, Maivi Nyström, Kyoko Oda, David J Pearce, Tommy Persson, **Alexandre Pichot**, Christian Sommer, **Yoshinori Tanabe**, Kenji Tei, Susumu Toriumi, Florian Wagner, Yoriko Yamamura

The members of the Honiden lab, and my friends in Japan and abroad

You

Extra slides

Novelty

- No existing label-based argument selection in Java (to the best of my knowledge)
- No existing combination of typestate and AI planning
- Query-based integration has similarities with aspect-oriented programming, but is fundamentally novel

Design

Implicit mutations

```
public class Socket {

    resource speed {
      properties @fast, @slow;
      int dataSpeed;

      void setFast() mutates this.speed:
        this: ++@fast. {
        dataSpeed = 100;
      }
      void setSlow() mutates this.speed:
        this: ++@slow. {
        dataSpeed = 10;
      }
    }
}
```

- Convenience feature: No need to declare "mutates this.x" if the method is declared inside the resource - this is implicit

Constrained fields

```
class MessageSender {

 resource state {
  properties @ready, @notReady;

  Socket s:((@ready)->(@open),(@notReady)->(@closed));

  void open() this: ++@ready. {
    s = new Socket();
    s.open(); //the final state of s is validated
  }
}

class Socket {
 resource state {
  properties @open, @closed;

  void open() this: ++@open. { ... }
 }
}
```

- **Field labels depend on owning object's labels**
- Implicitly always unique

The drop statement

```
class MessageSender {

 resource state {
  properties @ready, @notReady;

  Socket s:((@ready)->(@open),(@notReady)->(@closed));

  void close() this: -@ready. {
    s.close();
    drop @ready;
    s = new Socket();
  }
}

class Socket {
 resource state {
  properties @open, @closed;

  void Socket() result: ++@closed. {...}
 }
}
```

- **Explicitly delete labels of ‘this’**
  - Identify a precise point where a label is lost
- Relaxes expectations on constrained fields
- Possibly unnecessary??

Formalisation

The chaining operation

$$\Gamma \vdash (\mathsf{LS}_1, \rho_1) \oplus (\mathsf{LS}_2, \rho_2) \stackrel{\mathsf{def}}{=} ((\mathsf{LS}_+, \mathsf{LS}_=, \mathsf{LS}_-), \rho) \quad \text{where}$$

$$\mathsf{LS}_+ \stackrel{\mathsf{def}}{=} (\mathsf{rem}(\Gamma, \rho_2, \mathsf{LS}_1^+) \cup \mathsf{LS}_2^+)) \qquad \backslash(\mathsf{LS}_2^- \cup \mathsf{LS}_1^= \cup \mathsf{LS}_1^-)$$

$$etm \stackrel{\mathsf{def}}{=} \mathsf{sens}(\Gamma, \rho_1, \mathsf{LS}_2^= \setminus \mathsf{LS}_1^=) \cup \mathsf{sens}(\Gamma, \rho_2, \mathsf{LS}_1^= \setminus \mathsf{LS}_2^=)$$

$$\mathsf{LS}_= \stackrel{\mathsf{def}}{=} (\mathsf{LS}_1^= \cup \mathsf{LS}_2^=) \setminus etm \qquad \backslash(\mathsf{LS}_1^+ \cup \mathsf{LS}_2^-)$$

$$\mathsf{LS}_- \stackrel{\mathsf{def}}{=} (LS_2^- \cup LS_1^- \cup etm) \qquad \backslash \mathsf{LS}_1^+$$

$$\rho \stackrel{\mathsf{def}}{=} \rho_1 \cup \rho_2$$

Monday, 16 January 2012

Note: this has been slightly altered from the version in the thesis.
Discuss alternative notions of well-formedness

Disjunctive composition

$$(\mathsf{LS}_1, \rho_1) \otimes (\mathsf{LS}_2, \rho_2) \stackrel{\mathsf{def}}{=} ((\mathsf{LS}_1^+ \cap \mathsf{LS}_2^+,$$

$$\mathsf{LS}_1^= \cap \mathsf{LS}_2^=,$$

$$(\mathsf{LS}_1^- \cup \mathsf{LS}_2^- \cup (\mathsf{LS}_2^= \setminus \mathsf{LS}_1^=) \cup (\mathsf{LS}_1^= \setminus \mathsf{LS}_2^=)) \setminus (\mathsf{LS}_1^+ \cup \mathsf{LS}_2^+)$$

Property/resource polymorphism

```
class Base {
 resource r {
  properties @p;
  int i;
  void makeP() this: ++@p. {
    i = 0;
  } }
}

class E1 extends Base {
 resource r {
   int j;
   void makeP() this: ++@p. {
     i = 0;
     j = 0; //stronger def.
   } }
}

class E2 extends Base {
 resource r {
  String x;
  void makeP() this: ++@p. {
   x = “”; //different def.
  } }
}
```

- Overriding resources can add more state, more properties
- Overriding properties can redefine
  - Internal predicate
  - Temporal constraints (within limits)
- Properties cannot be moved to a different resource

Prior/posterior expanded signatures

```
class MessageSender {

 resource state {
  properties @ready, @notReady;

  Socket s:((@ready)->(@open),
(@notReady)->(@closed));

  void open() this: ++@ready. {
    s = new Socket();
    s.open(); //the final state of s
is validated
  }
}
```

`open() prior: (this: {}, this.s: {})`
`open() posterior: (this: {@ready}, this.s:{@open})`

- Full specification of the state of a method before and after execution
- Domain: fields in **this**, arguments, receiver (same as LS)
  - Note: in general, mutations are **only** permitted on these expressions

One exception is if we have a “fresh” expression. Mutations on these are always permitted.

3 resource access levels

```
class Demo {

 resource r {
  properties @a, @b;
  int x;

  //m in raw mode because of ++@a
  void m() this: ++@a, @b -@c. {
    x = 0; //@a and @b not checked
  }

  //m2 not in raw mode
  void m2() this: +@a, @b, -@c. {
   m(); //@a, @b, @c are checked
  }
 } //end of resource r

  //m3 has no access to r
  void m3() this: +@a, @b, -@c. {
    m(); //invalid because of -@c
  }

}
```

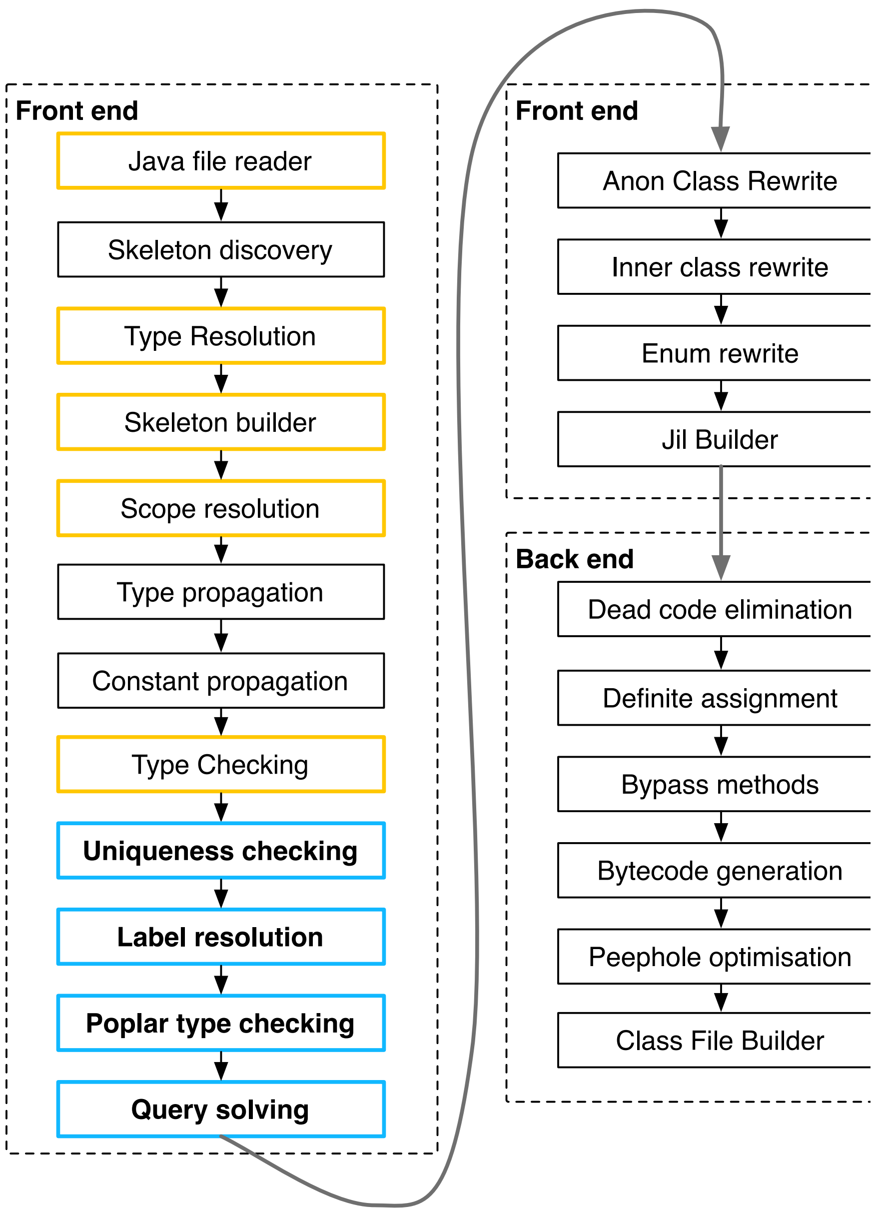
- **None** (weakest)
- **Mutates**
  - Can destroy properties
- **Raw** (with ++@p) (strongest)
  - Can write data directly in resource
  - ++ and = (invariants) are unchecked

Benefits of the resource/property model

- The structure of resources, in terms of properties and their relations, can often change without disturbing method contracts
- **Natural fit for AI planning algorithms**
- A "state" is a set of labels
    - **Client queries can match on a subset**

Implementation

Design decision: where to insert new stages?



- Early stage: Java classes remain very close to source code form, weak invariants provided and expected
- Late stage: compilation almost finished, strong invariants provided and expected
- Our new stages are inserted at a middle point, after Java type checking has been done

JKit

- Java compiler for research purposes, by David J Pearce
- Chosen as a foundation because it:
  - Compiles Java 5 (almost) fully
  - Is relatively recent
  - Has a straightforward design
- Written in Java

Integration link verification (future)

A straightforward implementation strategy:

- Store information about Poplar signatures in Java class files as *class file attributes* (standard feature)
  - In **client** classes, store assumptions about service side method contracts
  - In **service** classes, store the provided contracts
- To verify a link, simply check these assumptions against each other (using the "valid overriding" relation)

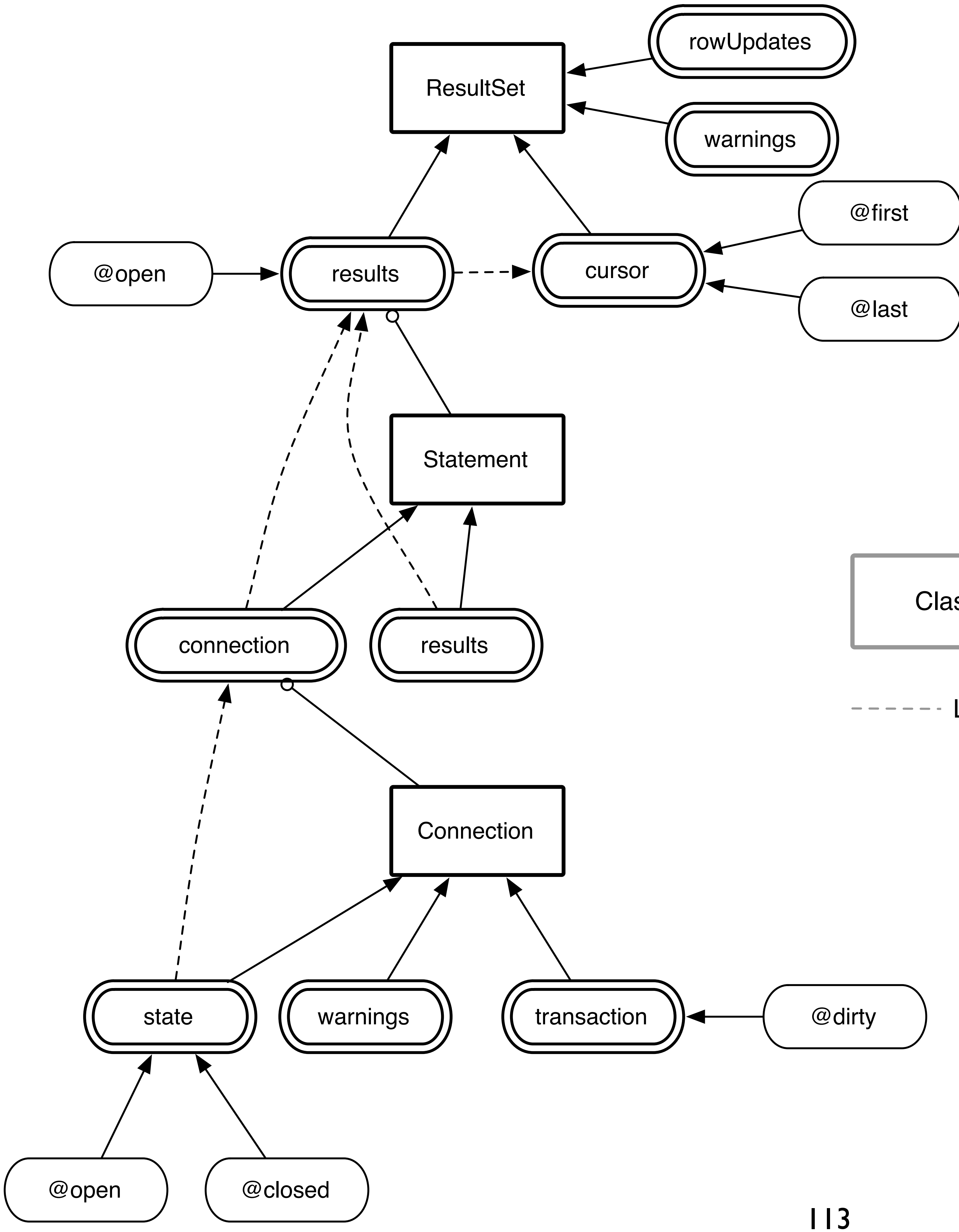
Conclusion

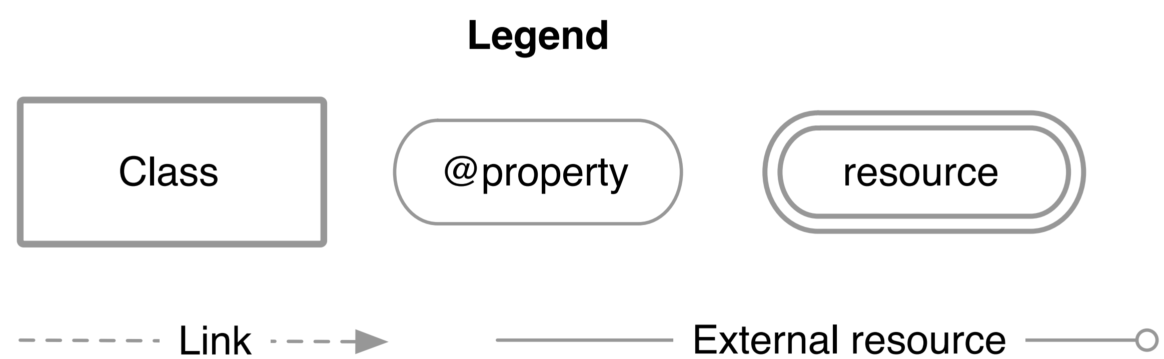
Resource links and external resources (future?)

```
class ItemList {
    resource list {
       properties @empty, @full;
       link ext[Item].hosted;
       List<Item> data;
    }
    resource[Item] hosted {
       properties @inList;
    }

    void add(Item i) mutates list:
       i: ++@inList. {
      data.add(i);
    }
    void empty() mutates list,
     any(Item).ext[ItemList].hosted:
     this: ++@empty. {
     data.removeAll();
    }
}
```

- **External resource:** one class provides properties for another class
- **Link:** mutation of x would implicitly also be a mutation of ext[D].hosted
- Limitation: we cannot automatically identify the external object that is operated on

Modelling JDBC with resource links





Drafts

```
public class Socket {
    resource state {
       properties @raw, @bound, @open, @closed;

       String remoteHost;
       boolean isConnected = false;
       int connectionSpeed = 0;

       Socket() this: ++@raw. { }

       void bind(SocketAddress bindPoint) this: -@raw, ++@bound. { }

       void connect() this: -@bound, ++@open. { }

       void send(byte[] data) this: @open; data: ++sentData. { }

       void receive(byte[] data, int offset, int max) this: open; offset:
receiveOffset; max: receiveMaxlen;
       data: ++receivedData. { }

       void close () this:-@open, ++@closed. { }

       void printInformation() this: @open. {
          println("Connected to " + remoteHost.toString() + " at " +
connectionSpeed + " kB/s");
       }
    }
}
```