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


Note that without typestate, there is no way to statically check that clients use temporal constraints correctly
Semantic assumptions (method contract)

```java
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

---

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”\[2\]

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?

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


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

• Prospector\textsuperscript{[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

Related: labelled argument selection

• Some languages (e.g. Lisp, ADA) allow for argument reordering and omission based on labels

• Labelled lambda calculus\textsuperscript{[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?

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**

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
Example - socket client

V. 1

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

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

V. 2

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

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

- 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

```java
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)
```
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

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**)

Monday, 16 January 2012
Produce-queries

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

Generate and substitute

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

```java
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)
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\(^{[1,2]}\)

Uniqueness kinds

<table>
<thead>
<tr>
<th>Kind</th>
<th>Assumption</th>
<th>Guarantee</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>None (may be aliased)</td>
<td>None</td>
</tr>
<tr>
<td>Unique</td>
<td>Is unique</td>
<td>Remains unique</td>
</tr>
<tr>
<td>Maintain</td>
<td>None (may be aliased)</td>
<td>Remains unique</td>
</tr>
</tbody>
</table>

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

```java
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.
<table>
<thead>
<tr>
<th></th>
<th>System</th>
<th>Poplar</th>
<th>B/G Effect</th>
<th>Typestate/Fugue</th>
<th>Labelled LC</th>
<th>Prospector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polymorphic regions</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subregions</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect summaries</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temporal state names</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Labelled arg. selection</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>State for individual frames</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Type-based queries</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Search/AI planning</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Unique pointers</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Static checking</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

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

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

• Introduction
• Design
• Demo
• Formalisation
• Implementation
• Case study and evaluation
• Conclusion
• Core calculus for an imperative fragment of Java\textsuperscript{[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

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 \texttt{++t} or \texttt{++@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
• 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

- 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\cite{1} track states in each frame independently

```java
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. {
        
    }
}
```

\[\Delta; \Gamma \vdash s : \tau + \text{LS}!\rho\]

\((\text{LS} \overset{\text{def}}{=} (\text{LS}_+, \text{LS}_\equiv, \text{LS}_-))\)

\[\Delta; \Gamma \vdash e!\emptyset \text{ ok}\]
\[\Delta; \Gamma \vdash x!\overline{\tau} \text{ ok}\]
\[\Delta; \Gamma \vdash \text{this}!\overline{\tau} \text{ ok}\]
\[\Delta; \Gamma \vdash (C)e!\overline{\tau} \text{ ok}\]

\[\Delta; \Gamma \vdash e : C : \text{Fresh}\]
\[\Delta; \Gamma \vdash e!\overline{\tau} \text{ ok}\]

\[U \rightsquigarrow U\]
Normal \rightsquigarrow Maintain
Unique \rightsquigarrow Maintain
Fresh \rightsquigarrow U

Uniqueness

\text{TS-SEQVARWRITE}

\[\Delta; \Gamma \vdash x = e; : \text{void} + \text{LS}_1!\rho_1\]
\[\Delta; \Gamma \vdash s_2 \ldots s_n : \tau + \text{LS}_2!\rho_2\]
\[\Delta; \Gamma \vdash e!\rho_2(x) \text{ ok}\]
\[\Delta; \Gamma \vdash e : C : U\]

\[(\text{LS}_1, \rho_1) \oplus (lflow(\text{LS}_2, x, e), rflow(\rho_2, x, e, C, U)) = (\text{LS}, \rho)\]

\[\Delta; \Gamma \vdash x = e; s_2 \ldots s_n : \tau + \text{LS}!\rho\]
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
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

Compiler tasks

Both of the essential stages have been implemented.
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**

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
Case study

• We will study JFreeChart\textsuperscript{[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

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 JPanel(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.getContentPane(c);
      frame.setVisible(true);
   }
}
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);
}
5.1. CASE STUDY: REFACTORING JFREECHART

Client

```java
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

```java
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.
    { ... }
```
private static JFreeChart useFactoryIndirect(XYSeriesCollection dataSet) {
    String title = tChartTitle;
    PlotOrientation po = (PlotOrientation) PlotOrientation.VERTICAL;
    String f = (String) tXAxisLabel;
    String a = (String) tCategoryAxisLabel;
    PlotOrientation po1 = (PlotOrientation) VERTICAL;
    String f1 = (String) tXAxisLabel1;
    String a1 = (String) tCategoryAxisLabel1;
    JFreeChart chart = useFactoryIndirect(dataset);
    //etc.
    return chart;
}

public static JFreeChart createXYBarChart(String title, String xAxisLabel, String yAxisLabel, IntervalXYDataset dataSet, PlotOrientation orientation, boolean legend, boolean tooltips, boolean urls) {
    ChartClient client = produce(JFreeChart, tXYBarChart);
    result = ++tXYBarChart;
    //etc.
    return client;
}

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

```java
public class ChartParameters {

    // Tag

    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.
    }

    // etc.

    // 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

---

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

- The client is updated correctly without any manual changes
Converting parameters to state

- 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

```java
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);
```

Note that this partially ordered solution can be linearised in many different ways, which are all valid.
Configuring the factory manually

```java
class ChartClient {
    //...
    private static void configureFactory(ChartFactory cf) {
        cf.factoryConfigured = true;
        cf.setGenTooltips(true);
        cf.setGenUrls(false);
        cf.setReqLegend(true);
        cf.resetConfiguration(); //This line violates the method’s contract
    }
    //...
}
```

- 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
5.1. CASE STUDY: REFACTORING JFREECHART

```java
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);
```

- 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.

Figure 5.10: Configuring the factory explicitly. This method is invalid with respect to its specification, since it erases the `@factoryConfigured` property on the final line.

Figure 5.11: When we have removed the seeded violation in Figure 5.10, we find this solution. The method `configureFactory` is now used in place of the individual setter methods. Note that this partially ordered solution can be linearised in many different ways, which are all valid.
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**
## Evolution consequences

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

Monday, 16 January 2012
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])

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)


Poplar publications

• Rejected
  • ECOOP 2011, POPL 2011, ESOP 2012, ...
    • Many reviewers liked the general approach, but it was probably too early

• Accepted

• Planned
  • New paper about design, formalism (possibly CBSE, SPLASH, TSE)
Outline

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

- Field labels depend on owning object’s labels
- Implicitly always unique
The drop statement

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

```java
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. {...}
    }
}
```
Formalisation
The chaining operation

\[ \Gamma \vdash (\text{LS}_1, \rho_1) \oplus (\text{LS}_2, \rho_2) \overset{\text{def}}{=} ((\text{LS}_+, \text{LS}_-, \text{LS}_-), \rho) \quad \text{where} \]

\[ \text{LS}_+ \overset{\text{def}}{=} (\text{rem}(\Gamma, \rho_2, \text{LS}_1^+) \cup \text{LS}_2^+) \setminus (\text{LS}_2^- \cup \text{LS}_1^- \cup \text{LS}_1^-) \]

\[ e tm \overset{\text{def}}{=} \text{sens}(\Gamma, \rho_1, \text{LS}_2^\exists \setminus \text{LS}_2^\exists) \cup \text{sens}(\Gamma, \rho_2, \text{LS}_1^\exists \setminus \text{LS}_2^\exists) \]

\[ \text{LS}_- \overset{\text{def}}{=} (\text{LS}_1^\exists \cup \text{LS}_2^\exists) \setminus e tm \quad \setminus (\text{LS}_1^+ \cup \text{LS}_2^-) \]

\[ \text{LS}_- \overset{\text{def}}{=} (\text{LS}_2^- \cup \text{LS}_1^- \cup e tm) \quad \setminus \text{LS}_1^+ \]

\[ \rho \overset{\text{def}}{=} \rho_1 \cup \rho_2 \]

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

\[(LS_1, \rho_1) \otimes (LS_2, \rho_2) \overset{\text{def}}{=} ((LS_1^+ \cap LS_2^+, \quad \text{LS}_1^- \cap \text{LS}_2^-),

\quad (LS_1^- \cup \text{LS}_2^- \cup (\text{LS}_2^- \setminus \text{LS}_1^-) \cup (\text{LS}_1^- \setminus \text{LS}_2^-)) \setminus (\text{LS}_1^+ \cup \text{LS}_2^+)\]
Property/resource polymorphism

```java
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

• 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?)

```java
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");
        }
    }
}