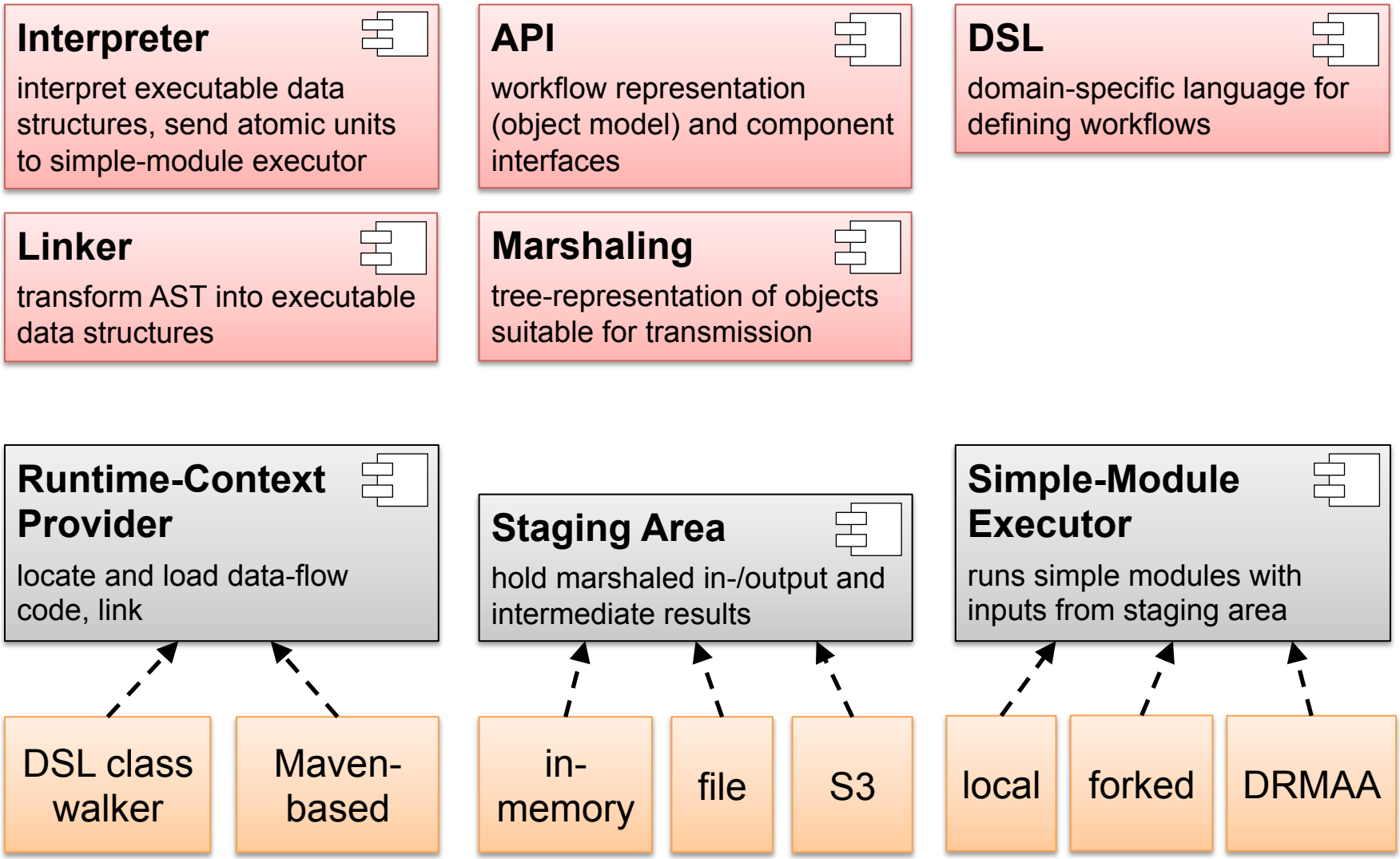
CloudKeeper Modularity

Architecture

Select Component Details

Component Diagram



Workflow-Execution Use Cases

	Execution Environment	Source Repository	Artifact Repository
Development			
Debugging	single JVM on laptop	not checked in	not checked in
Smoke Tests	multiple JVMs on laptop	"	not checked in or snapshot
Realistic Tests	cluster	"	snapshot
Production			
Real Data	"	checked in	release

Maven-based Runtime-Context Provider

CloudKeeper Bundle

- Logically: shared library
- Physically: Maven artifact generated by plugin
- Dependency resolution during runtime
- Dynamic class-loader creation

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bundle xmlns="http://www.svbio.com/cloudkeeper/1.0.0">
    <cloudkeeper-version>2.0.0.0-SNAPSHOT</cloudkeeper-version>
    <creation-time>2015-09-04T12:29:50.276-07:00</creation-time>
    <packages>
        <package>
            <qualified-name>com.svbio.cloudkeeper.samples.maven</qualified-name>
            <declarations>
                <simple-module-declaration>
                    <simple-name>AvgLineLengthModule</simple-name>
                    <annotations/>
                    <ports>
                        <in-port>
                            <name>text</name>
                            <annotations/>
                            <declared-type ref="java.lang.String"/>
```

Implementing a CloudKeeper Service

Simple API for Controlling Workflow Executions

```
MutableModule<?> module = new MutableProxyModule()
    .setDeclaration("com.svbio.test.PiModule");

WorkflowExecution workflowExecution = cloudKeeperEnvironment
    .newWorkflowExecutionBuilder(module)
    .setInputs(Collections.singletonMap(
        SimpleName.identifier("precision"), precision)
    )
    .setBundleIdentifiers(Collections.singletonList(Bundles.bundleIdentifierFromMaven(
        "com.svbio.ckmodules",
        "ckmodules-test",
        Version.valueOf("1.1.0.12-SNAPSHOT")
    )))
    .start();

String result = (String) WorkflowExecutions
    .getOutputValue(workflowExecution, "digits", 1, TimeUnit.MINUTES)
```

The CloudKeeper Data-Flow Programming Language

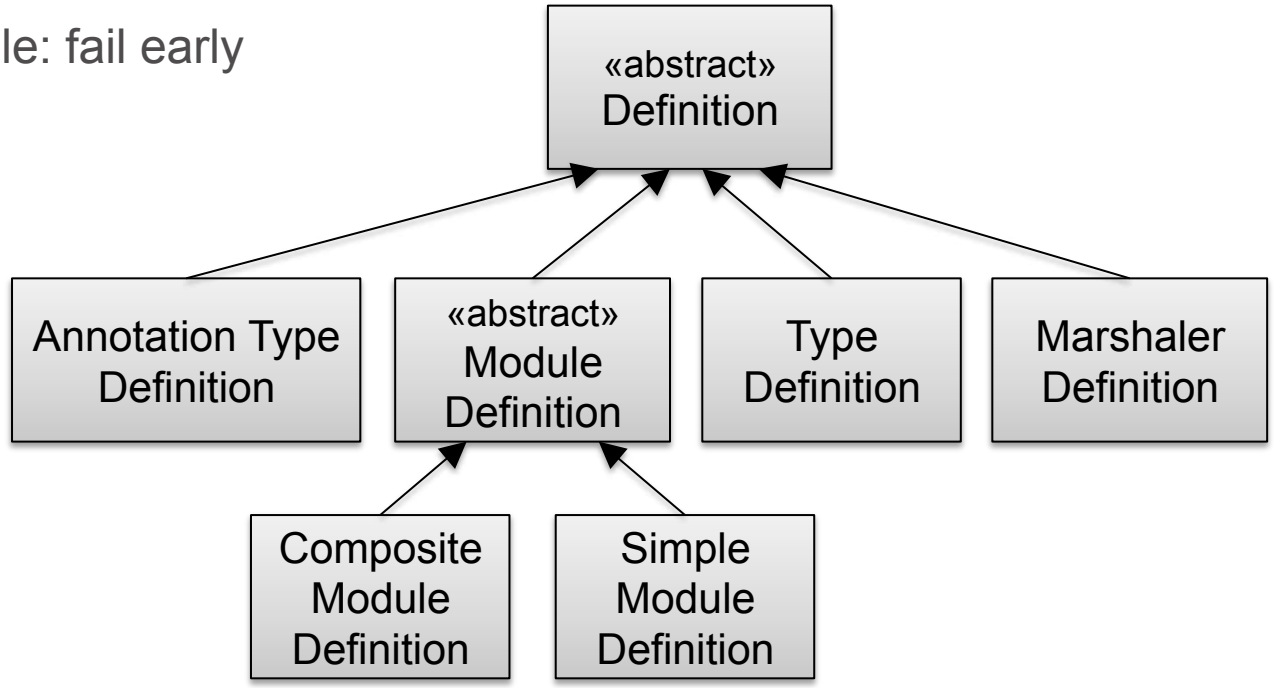
Fundamental Tasks: Compile, Link, Report Errors

Type System

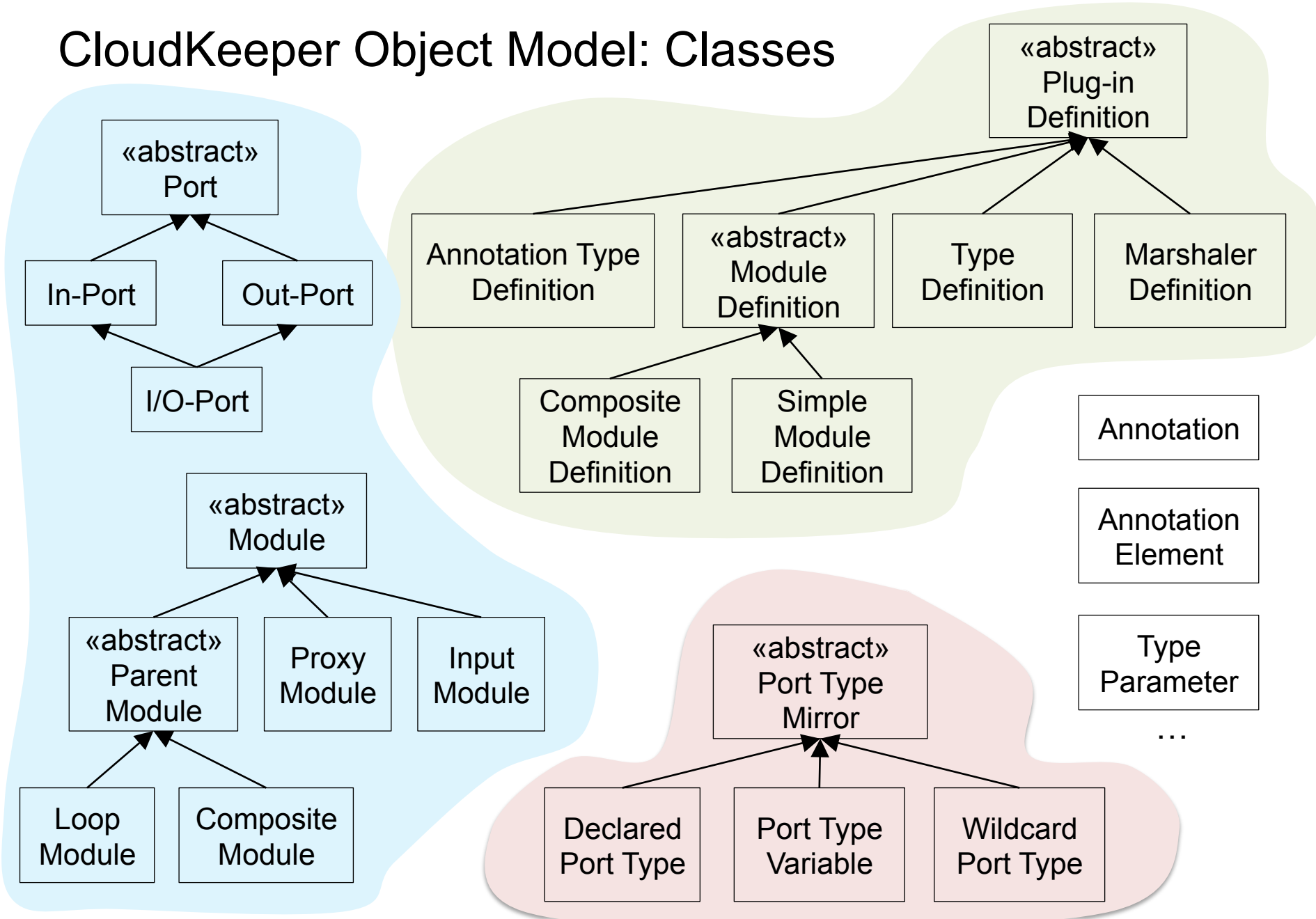
Basic Concepts

Compiled Language

- Every workflow linked against repository of definitions
  - eager linking
- Static typing
- Rationale: fail early



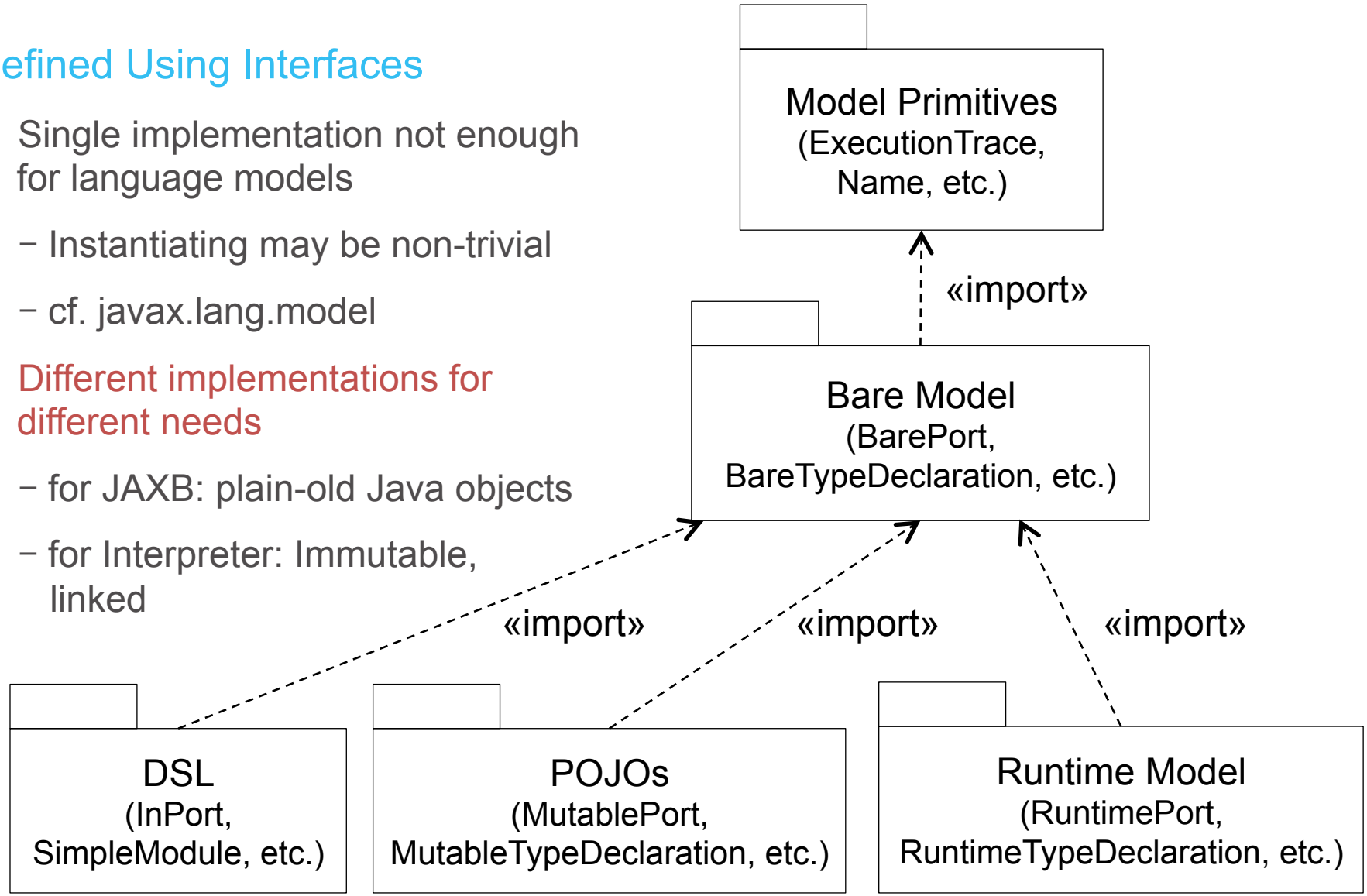
CloudKeeper Object Model: Classes



CloudKeeper Object Model: Packages

Defined Using Interfaces

- Single implementation not enough for language models
  - Instantiating may be non-trivial
  - cf. javax.lang.model
- Different implementations for different needs
  - for JAXB: plain-old Java objects
  - for Interpreter: Immutable, linked



CloudKeeper API for Defining Workflows

CloudKeeper POJO Classes

- Mutable representation of (bare) AST
- Allow programmatic definition of CloudKeeper modules

```
public abstract static class CompositeWithInput
        extends CompositeModule<CompositeWithInput> {
    public abstract InPort<Collection<Integer>> number();
    public abstract OutPort<Integer> list();

    InputModule<Integer> one = value(42);

    { list().from(one); }
}
```

```
new MutableCompositeModule()
    .setDeclarationName(CompositeWithInput.class.getName())
    .setDeclaredPorts(Arrays.asList(
        new MutableInPort()
            .setName("number")
            .setType(
                new MutableParameterizedPortType()
                    .setRawTypeName(Collection.class.getName())
                    .setActualTypeArguments(Arrays.asList(
                        new MutableLinkedTypeDeclaration()
                            .setName(Integer.class.getName())
                    ))
            ),
        new MutableOutPort()
            .setName("list")
            .setType(
                new MutableTypeDeclarationReference()
                    .setName(Integer.class.getName()
            )
    ))
// ...
```

XML Bindings for CloudKeeper Object Model

JAXB Annotations

- On Java Bean-style implementation of domain interfaces
- JAXB part of Java SE

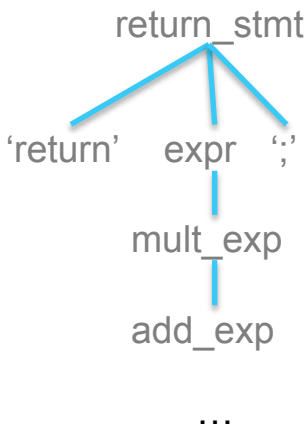
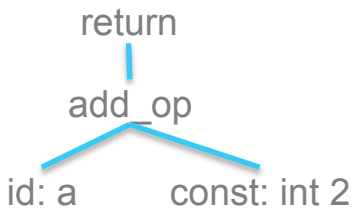
XML Schema Exists

- Reliable external interface – e.g., for XPath queries
- Immediate integration with IDEs

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <repository xmlns="http://www.svbio.com/cloudkeeper/1.0.0"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://www.svbio.com/cloudkeeper/1.0.0 http://www.florian-schoppmann.net/cloudkeeper-v1_0_0.xsd">
5      <bundleIdentifier>
6          <name>com.svbio.cloudkeeper.examples.bundles.simple</name>
7          <version>1.0.0-SNAPSHOT</version>
8          <locations>
9              <location>x-maven:com.svbio.cloudkeeper.examples.bundles:simple:ckbundle.zip:1.0.0-SNAPSHOT</location>
10         </locations>
11         <
12     </bu
13     <cre
14     <dec
15
16
17             <annotation ref="com.svbio.cloudkeeper.model.annotations.SimpleModulePlugin">
```

```
name          http://www.svbio.com/cloudkeeper/1.0.0
locations     http://www.svbio.com/cloudkeeper/1.0.0
version       http://www.svbio.com/cloudkeeper/1.0.0
Press ^Space to view tags from other namespaces >>
```

CloudKeeper Is a Programming Language!

Source Code	Java, Scala, etc.	CloudKeeper DSL, XML
Tokenization `[0-9]+`	JLS 8, §3 Lexical Structure	
Parse Tree 	JLS 8, §19 Syntax Tree representation of deriving start symbol	Process instances from host language
Abstract Syntax Tree 	syntactic representation of source code	
Executable	byte code (.class/.jar)	verified AST (.xml/.ckbundle)

Dynamic Linking: Java vs. CloudKeeper

Executable	byte code (e.g., .class file)	AST in memory (alternatively, .xml file)
Load Executables	on-demand when resolving symbolic references, no package management	up front by package manager
Resolve Symbolic References	by class loader (e.g., scan class path), resort to parent class loader, may trigger Load Executables	search “repository” consisting of “bundles” that contain definitions
Resolution Errors	thrown when class used	immediately – fail early
Verification and Initialization	correctness checks static initializer blocks, etc.	preprocessing

The Java Type System

Convenient, But not Ideal

- No covariant type parameters

```
List<Number>  :>  ArrayList<Integer>
```

```
ArrayList<Integer> arrayList = new ArrayList<>();
List<Number> list = arrayList;  // Not legal, but suppose it was
list.add(3.0);
```

- Java solution: wildcards and type bounds

```
ArrayList<Integer> arrayList = new ArrayList<>();
List<? extends Number> list = arrayList;  // Now legal
list.add(3.0);  // This is now illegal
```

- CloudKeeper port types are immutable – problem would not arise!
  - Wildcards create unnecessary visual clutter

Error Reporting

DSL Debug Information is Preserved

- Keeps record of Java source file and line number
- Linking failures produce “linking backtrace”
  - Logical containment chain

```
public abstract class MissingMergeModule
        extends CompositeModule<MissingMergeModule> {
    public abstract InPort<Collection<Integer>> inArrayPort();
    public abstract OutPort<Integer> outPort();

    Sum sum = child(Sum.class).
        firstPort().from(forEach(inArrayPort())).
        secondPort().from(value(1));

    { outPort().from(sum.outPort()); }
}
```

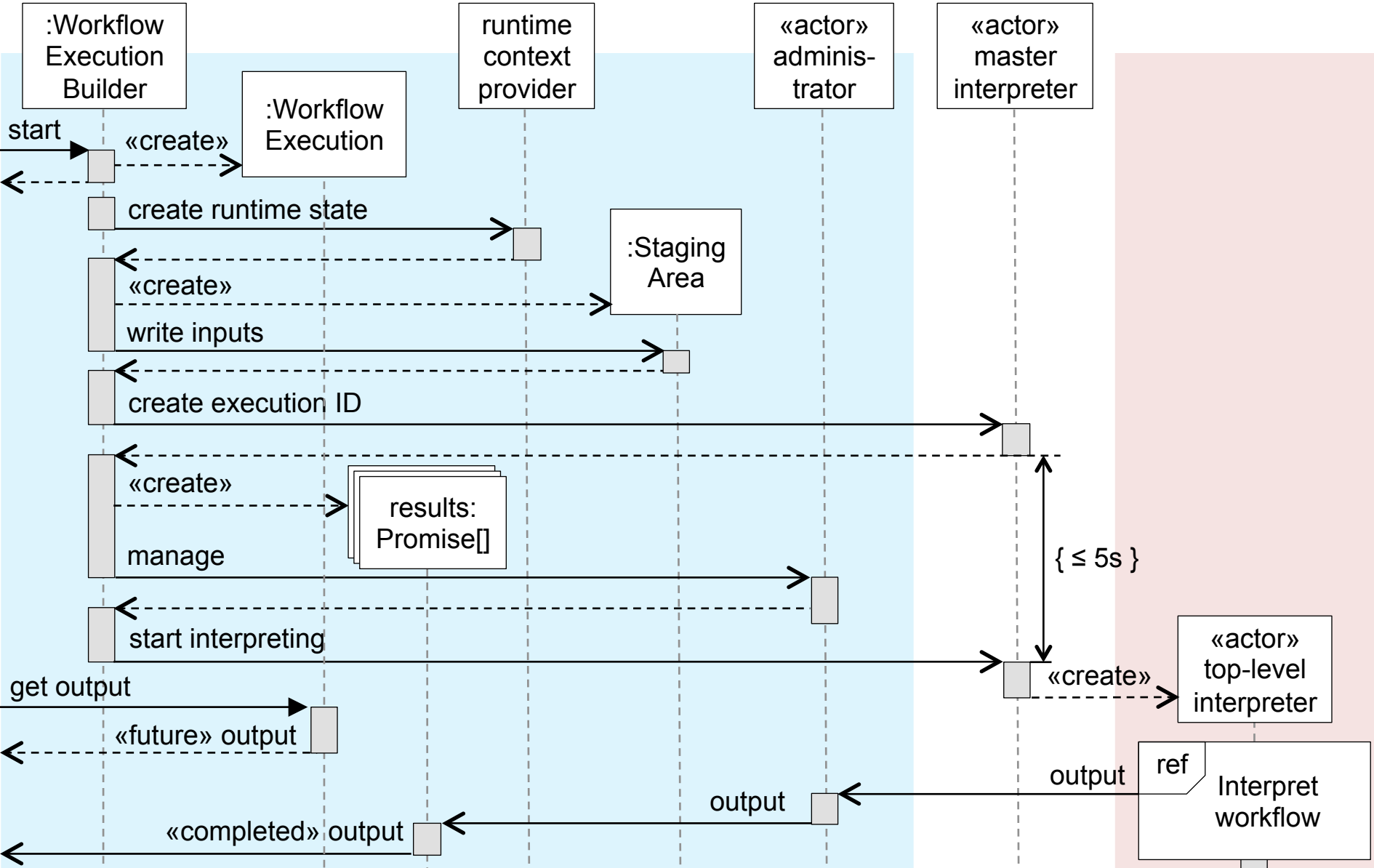
```
com.svbio.cloudkeeper.linker.ConstraintException: Connection from out-port outPort in composite module sum to
out-port outPort in composite module null is not a combine-into-array connection. Outgoing connections from
out-ports of an apply-to-all module must be combine-into-array connections.

Linking backtrace:
    connection sum#outPort -> null#outPort; MissingMergeModule.<init>(MissingMergeModule.java:19)
    composite module null; NoMergeTest.missingMergeTest(NoMergeConnectionTest.java:29)
```

The CloudKeeper Interpreter

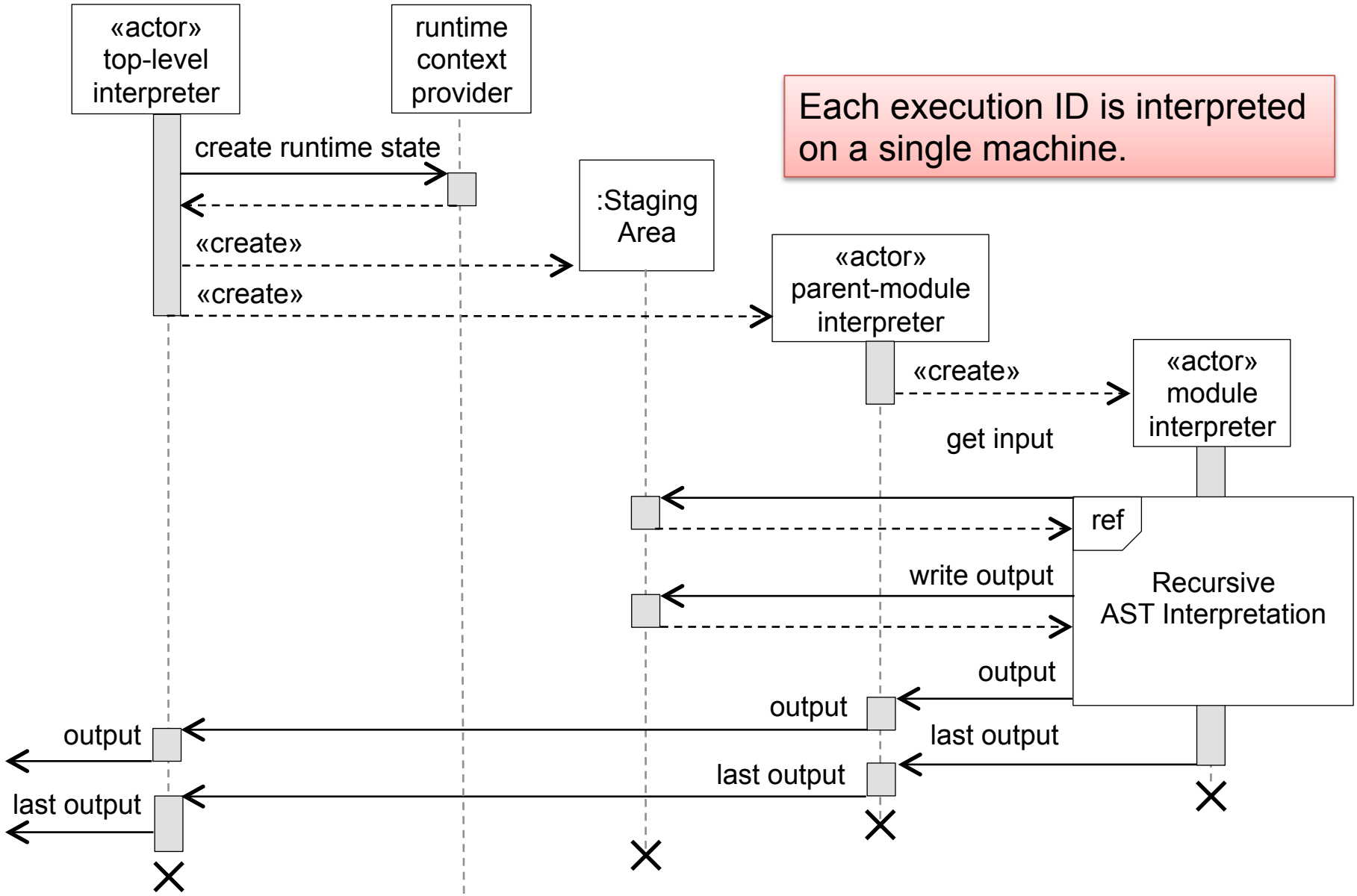
Scalability

Computing a Consistent Resume State

High-Level Components Involved in Starting Executions



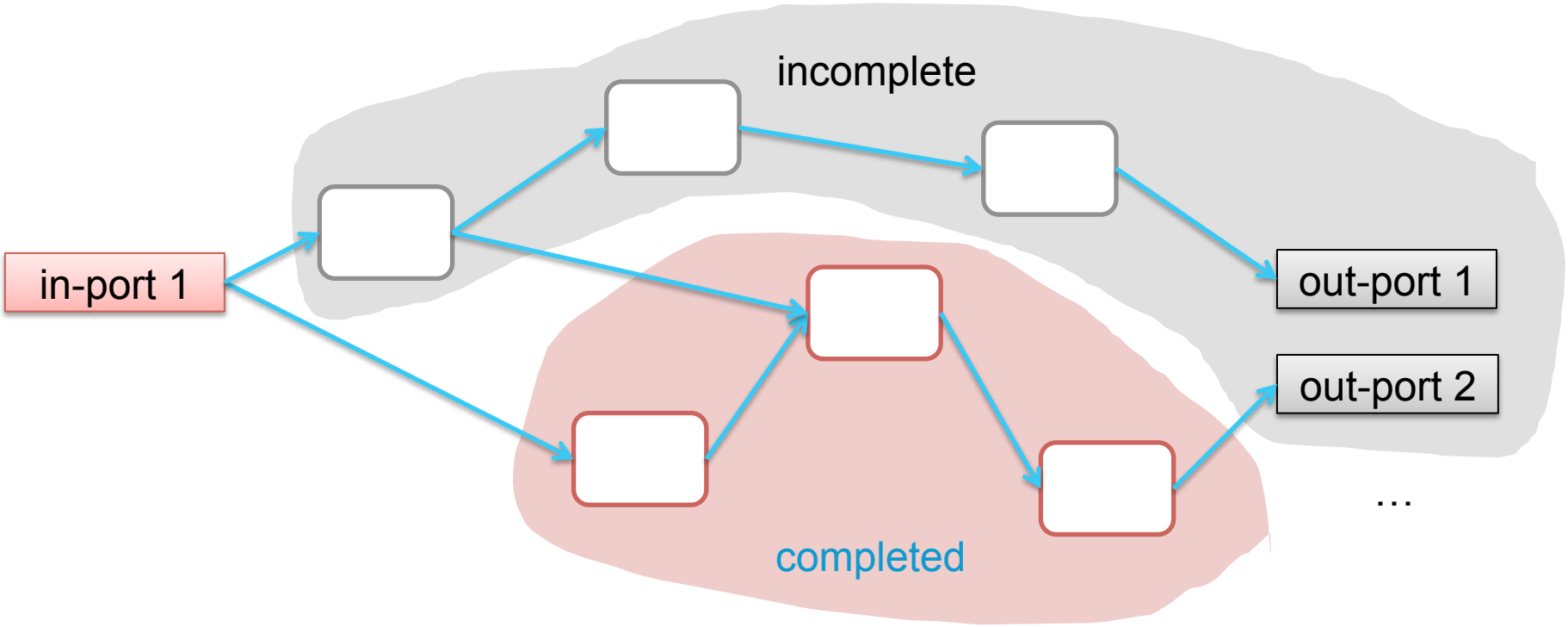
Interpreting Workflows



Restarting Workflows (1/3)

Recompute as little as possible – but as much as necessary

- Restarting should not impact set of possible results
  - there is linear order of module executions with same results
- Must invalidate successors of non-deterministic modules

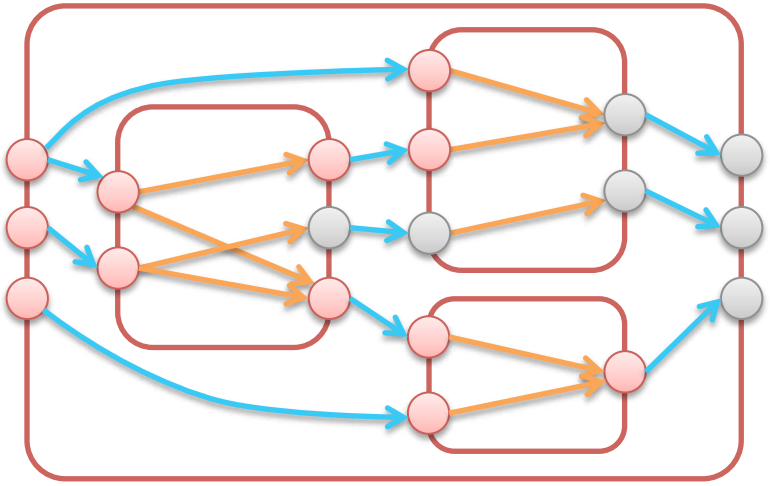


Restarting Workflows (2/3)

Requirements

- Single source of truth: the staging area
  - No transaction log necessary
- Motivation: Loose coupling, encapsulation, avoid unnecessary dependencies, etc.
- Robustness with respect to missing values

How to reconstruct execution state?



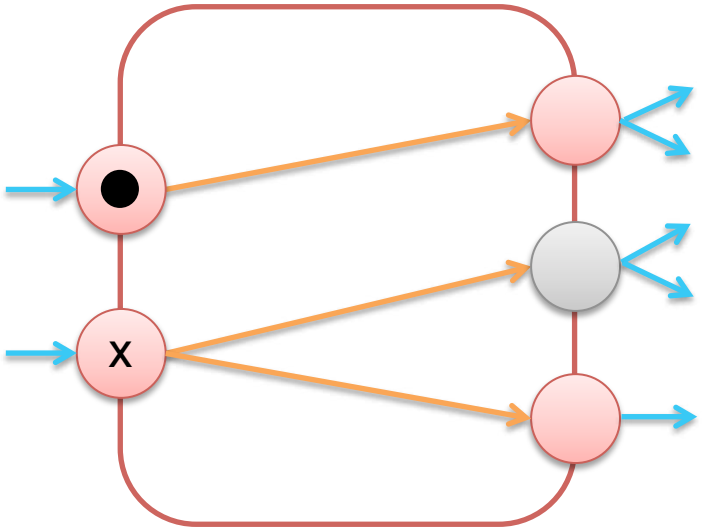
has value

no value

Restarting Workflows (3/3)

Main Problem

- Find “boundary” of ports so that when triggered:
  - All needed out-port will be computed
  - No port will receive value more than once
  - Minimal number of recomputed modules

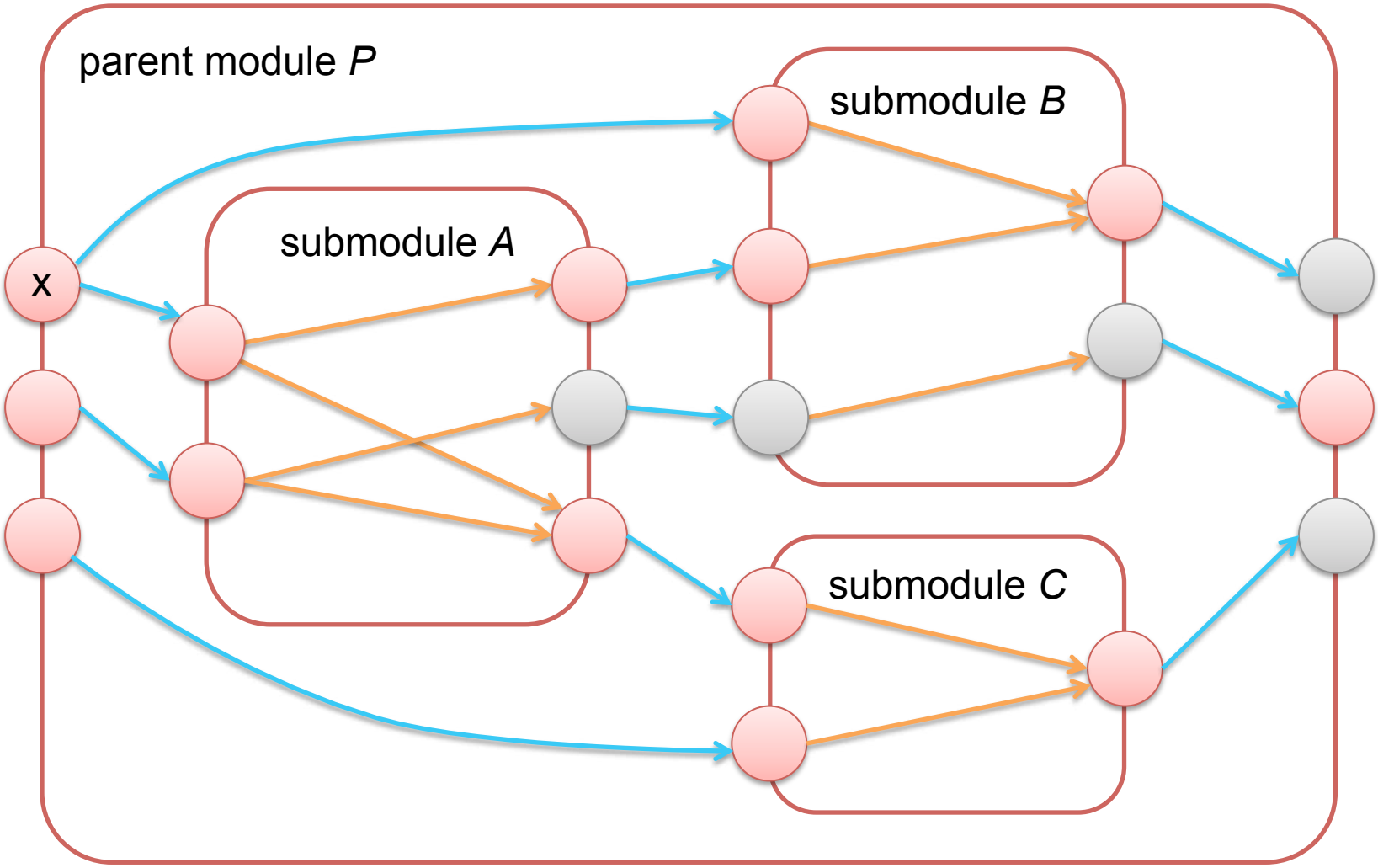


Trigger port

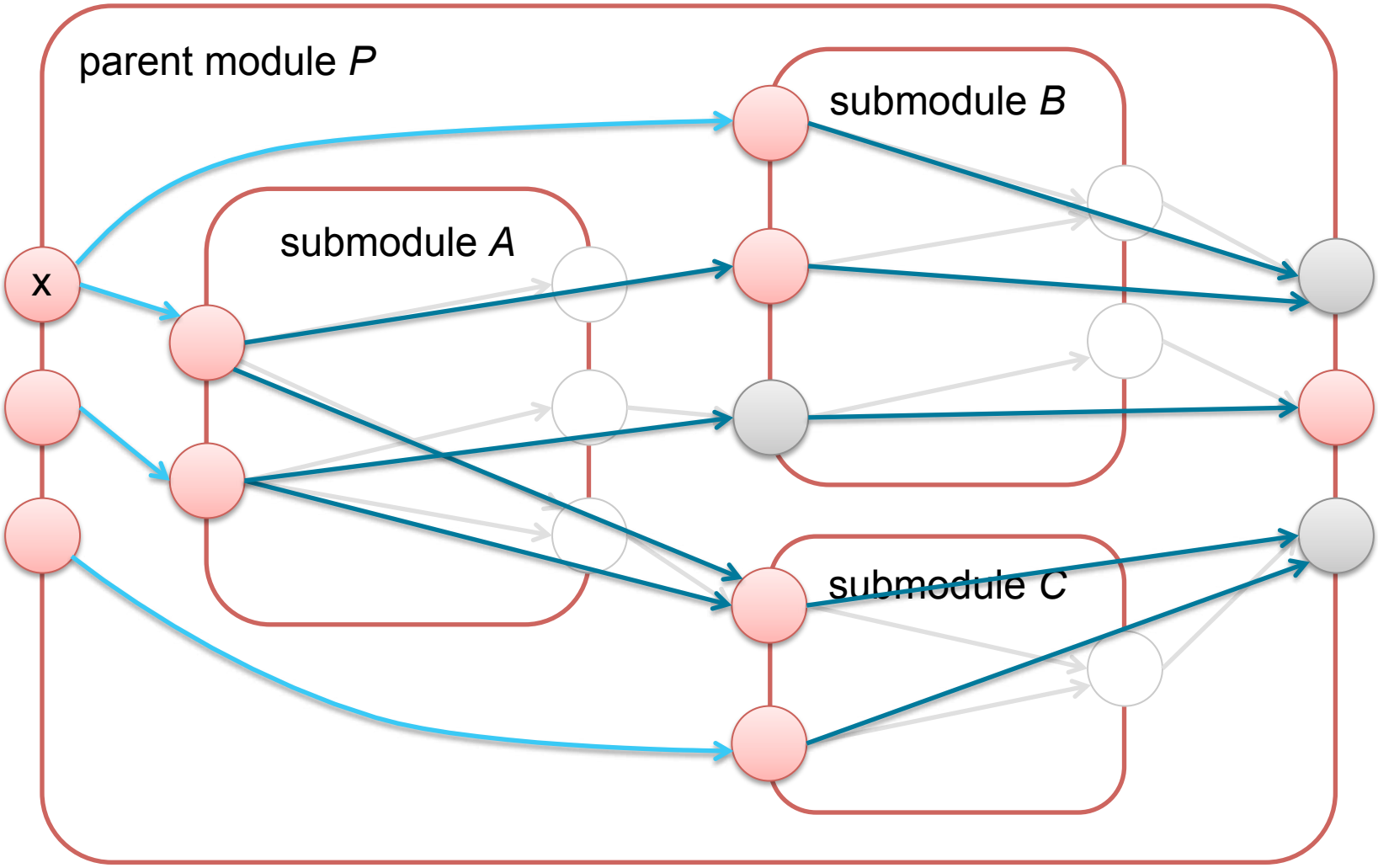
Do not trigger, will receive new value

Do not trigger, irrelevant

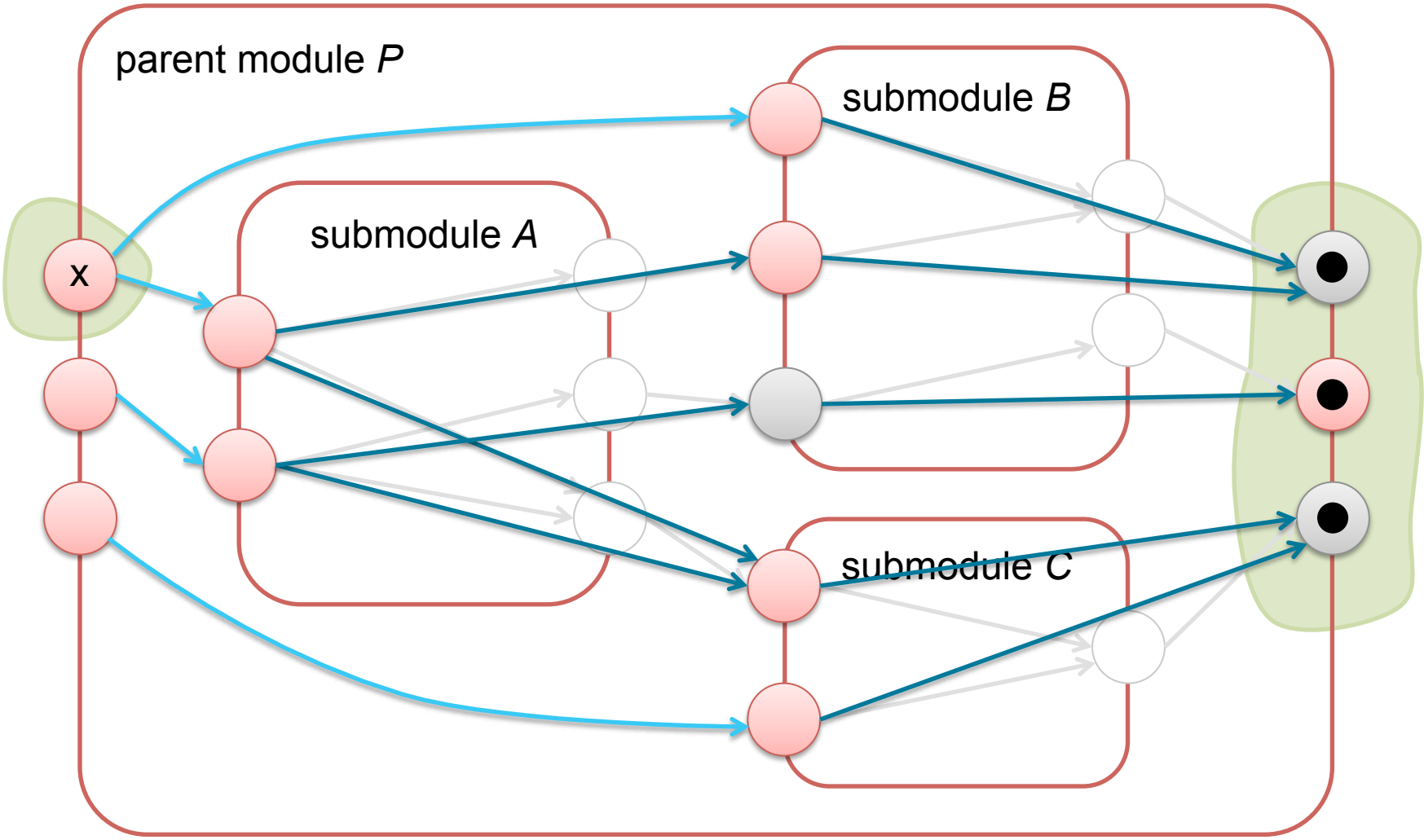
Restarting Workflows, Dependency Graph



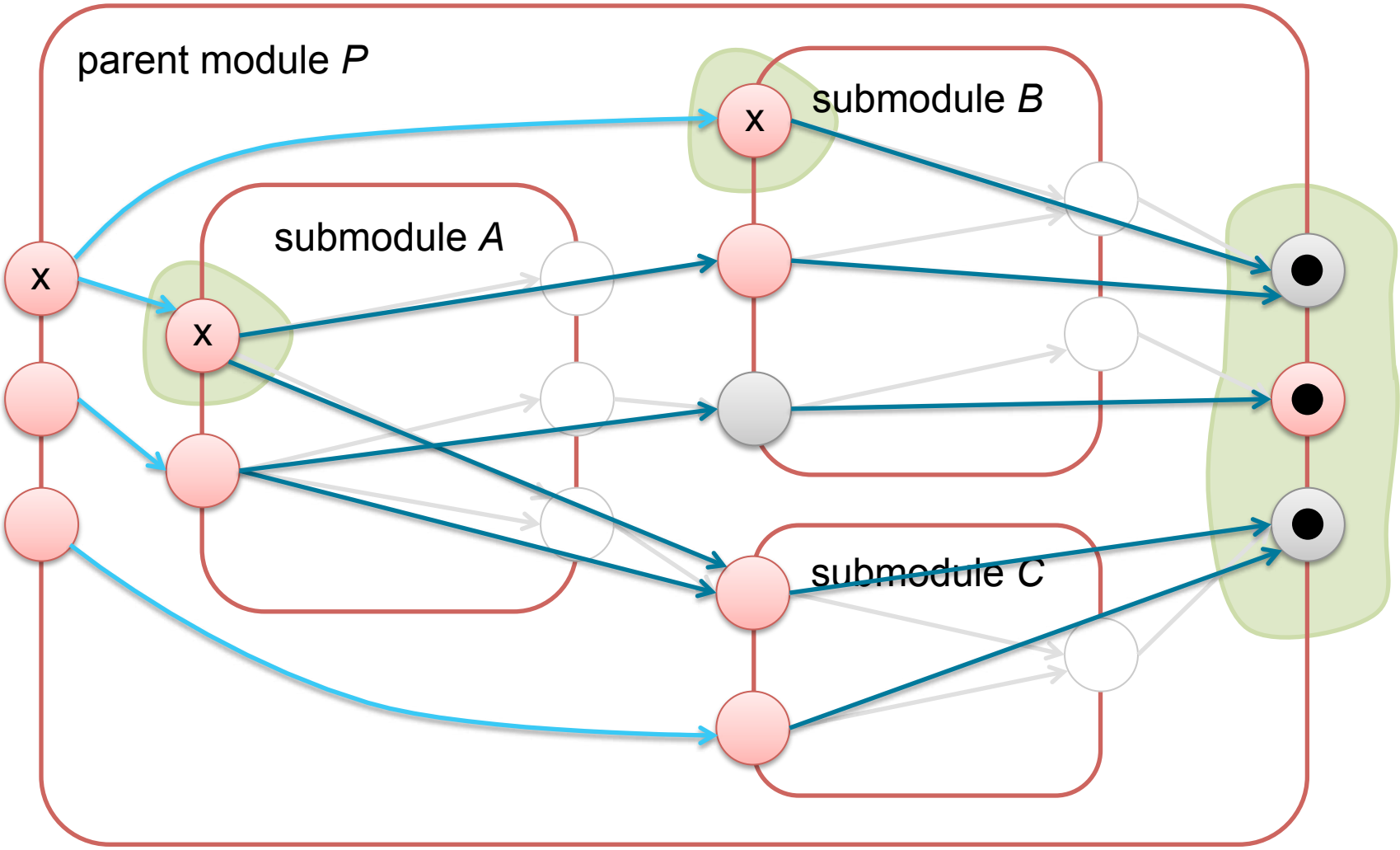
Restarting Workflows, Dependency Graph



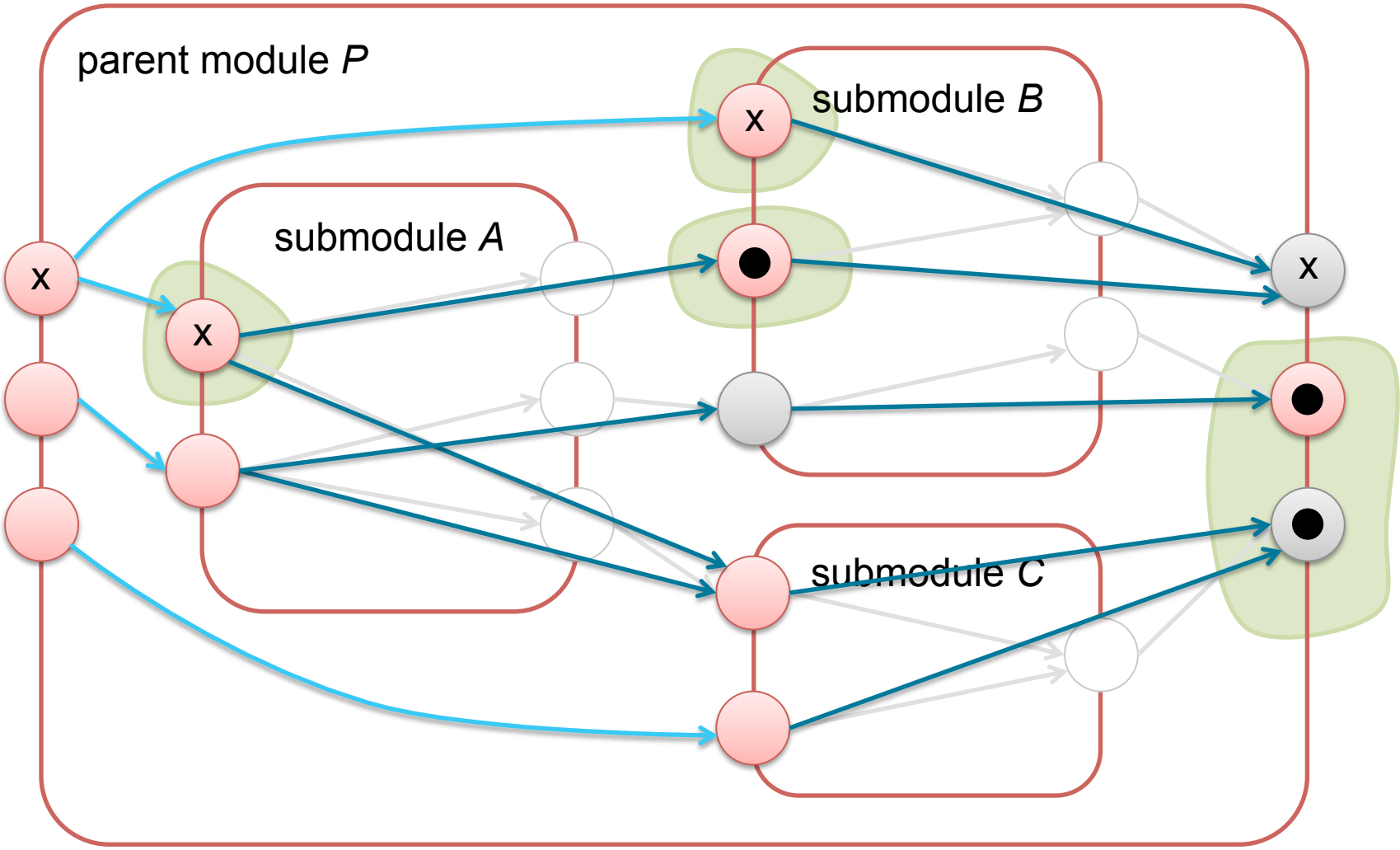
Restarting Workflows, Dependency Graph



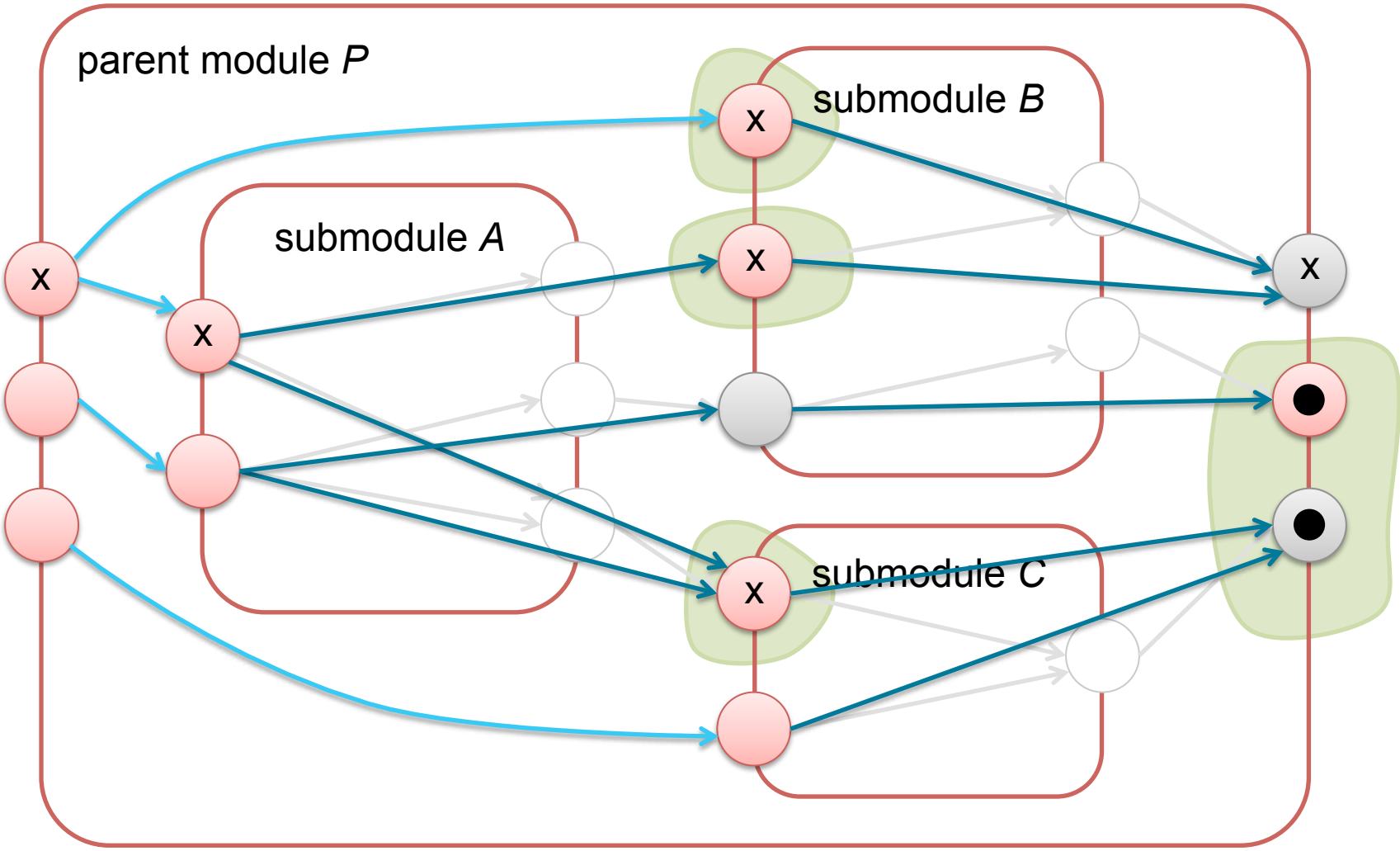
Restarting Workflows, Dependency Graph



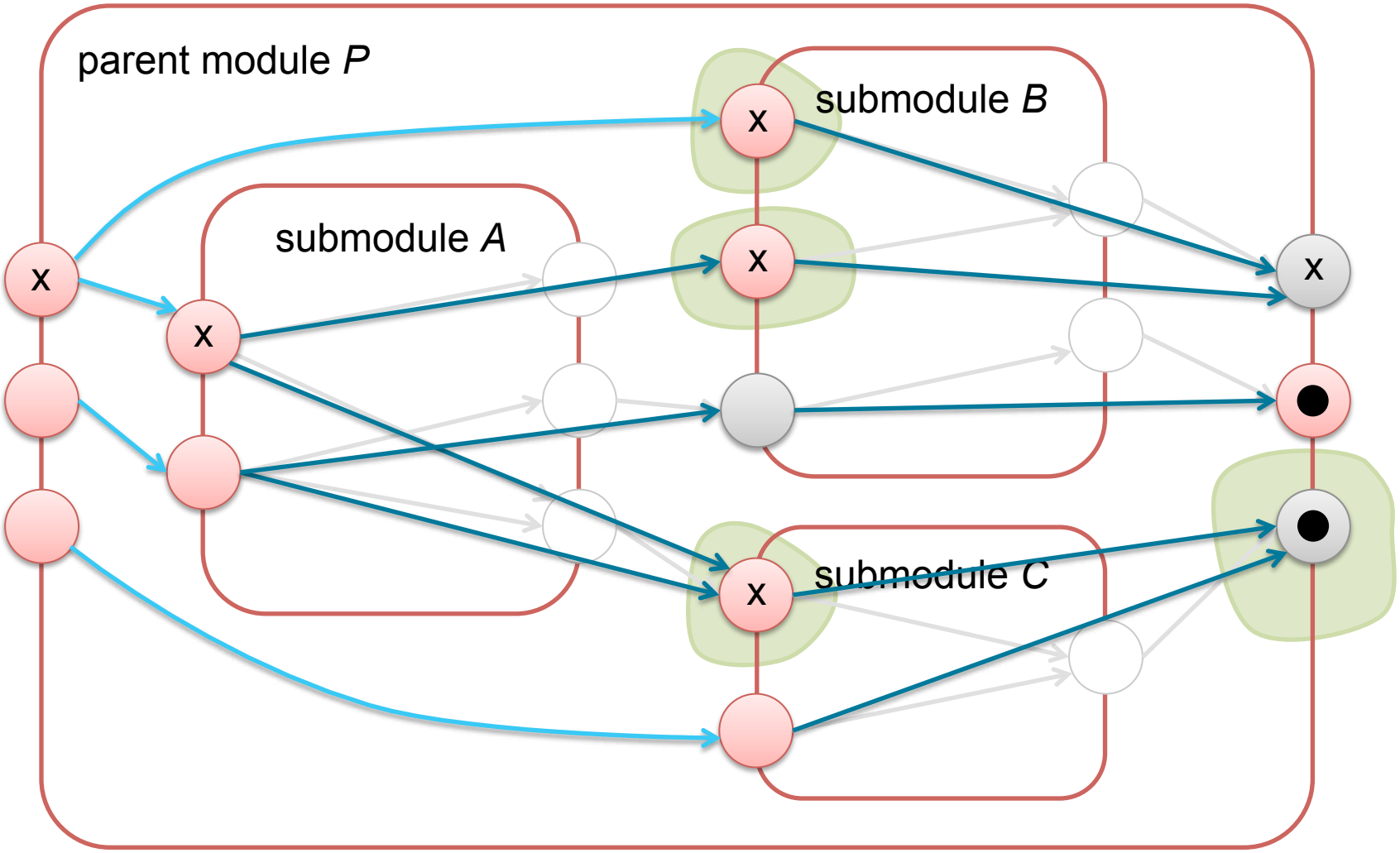
Restarting Workflows, Dependency Graph



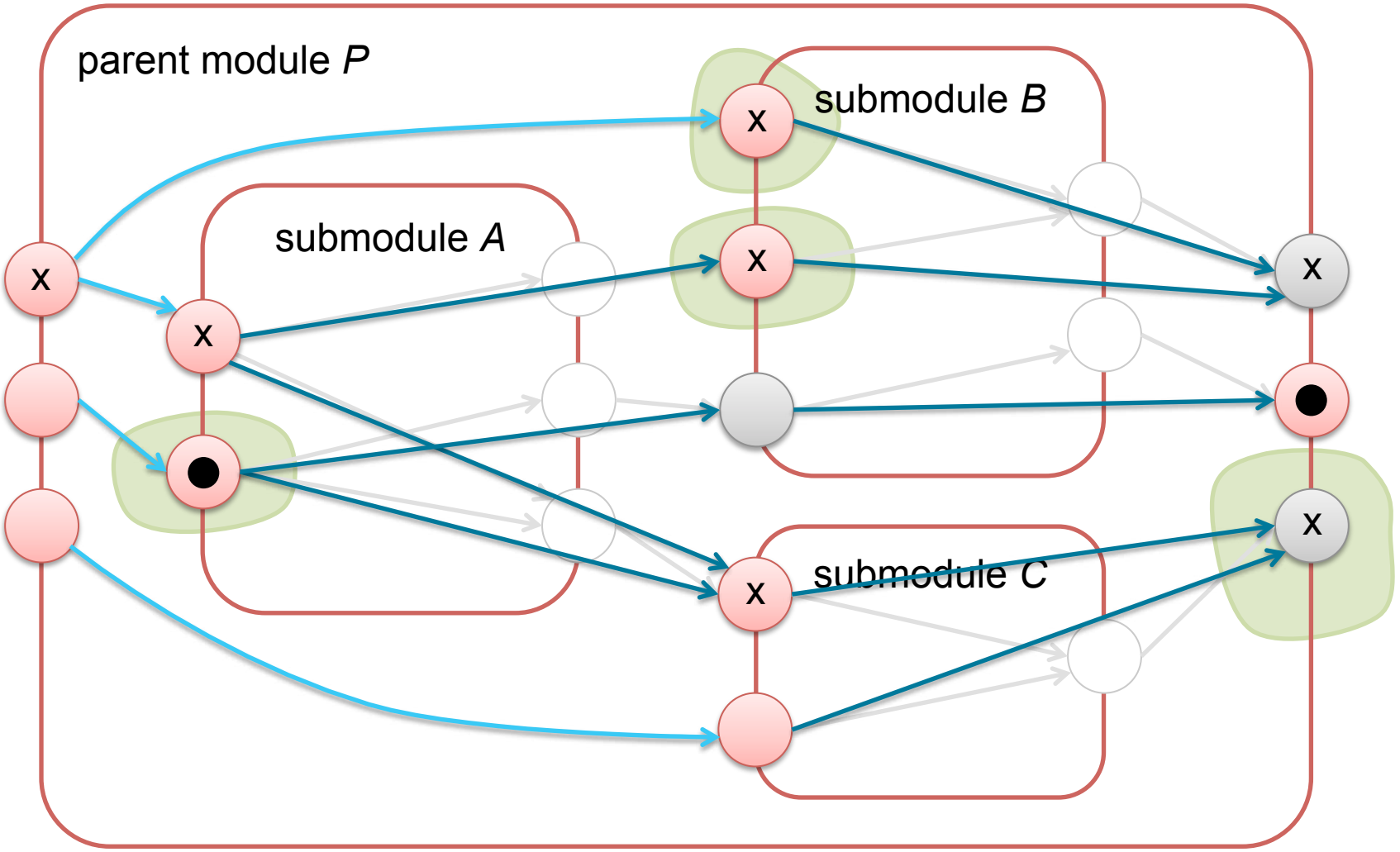
Restarting Workflows, Dependency Graph



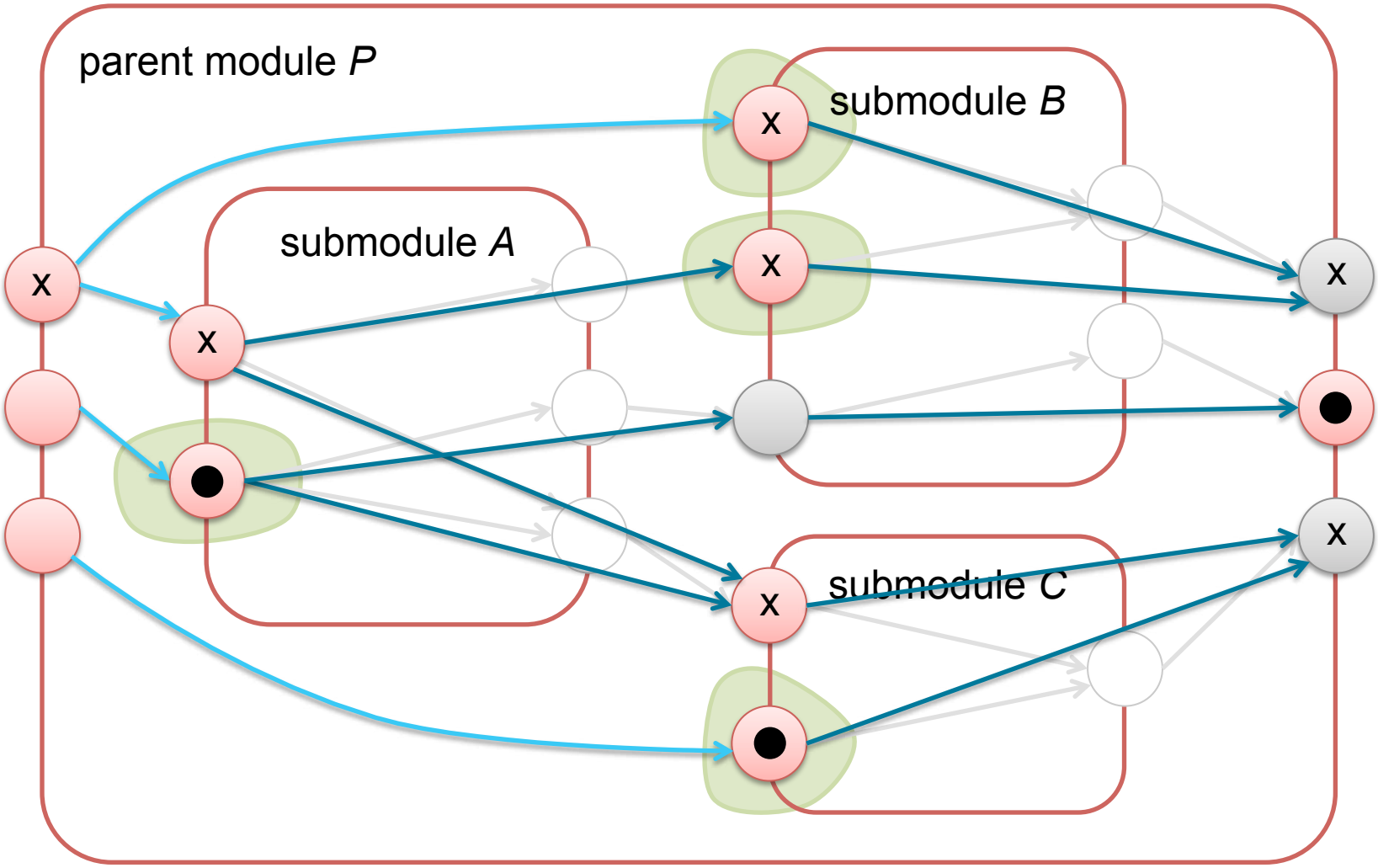
Restarting Workflows, Dependency Graph



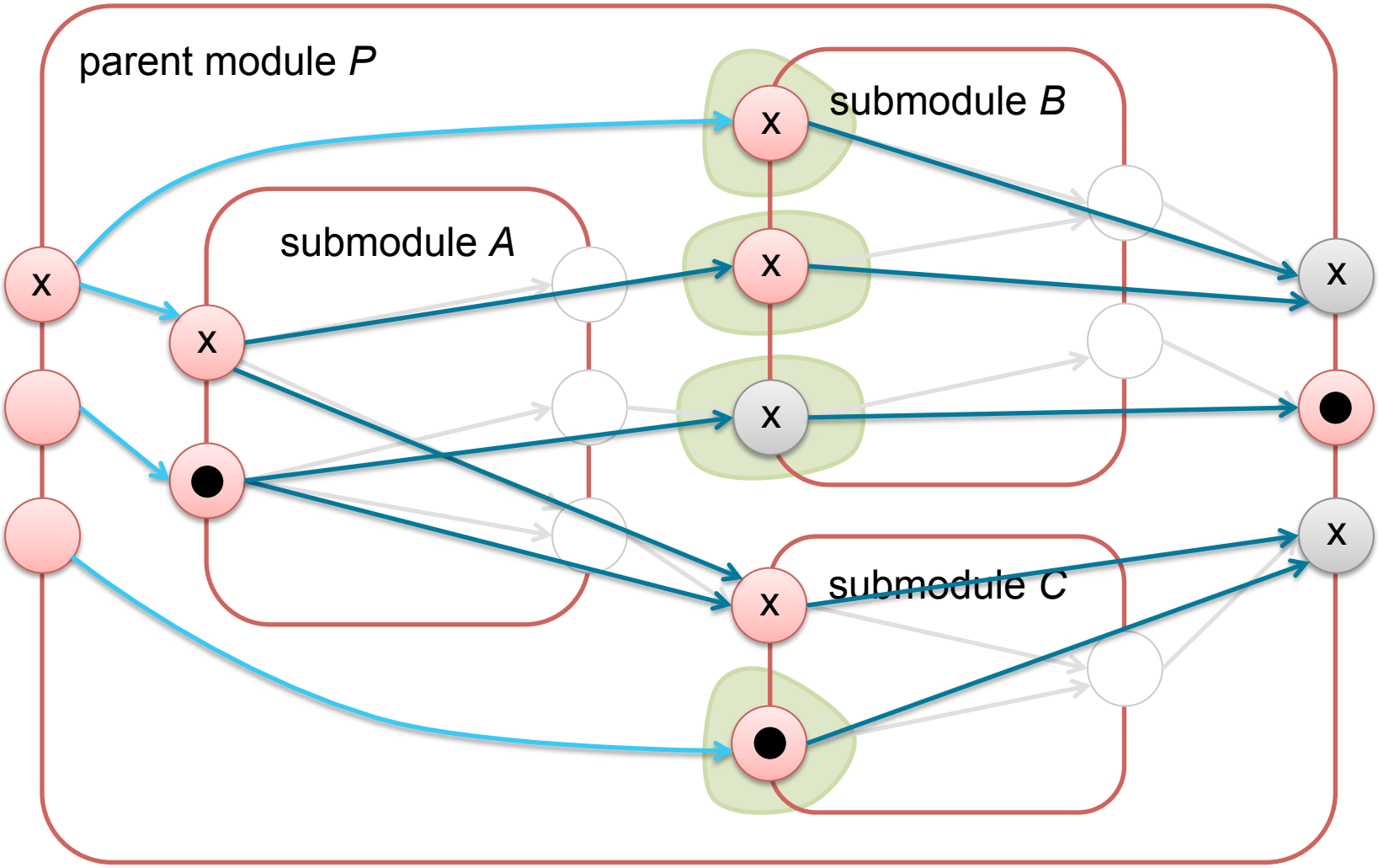
Restarting Workflows, Dependency Graph



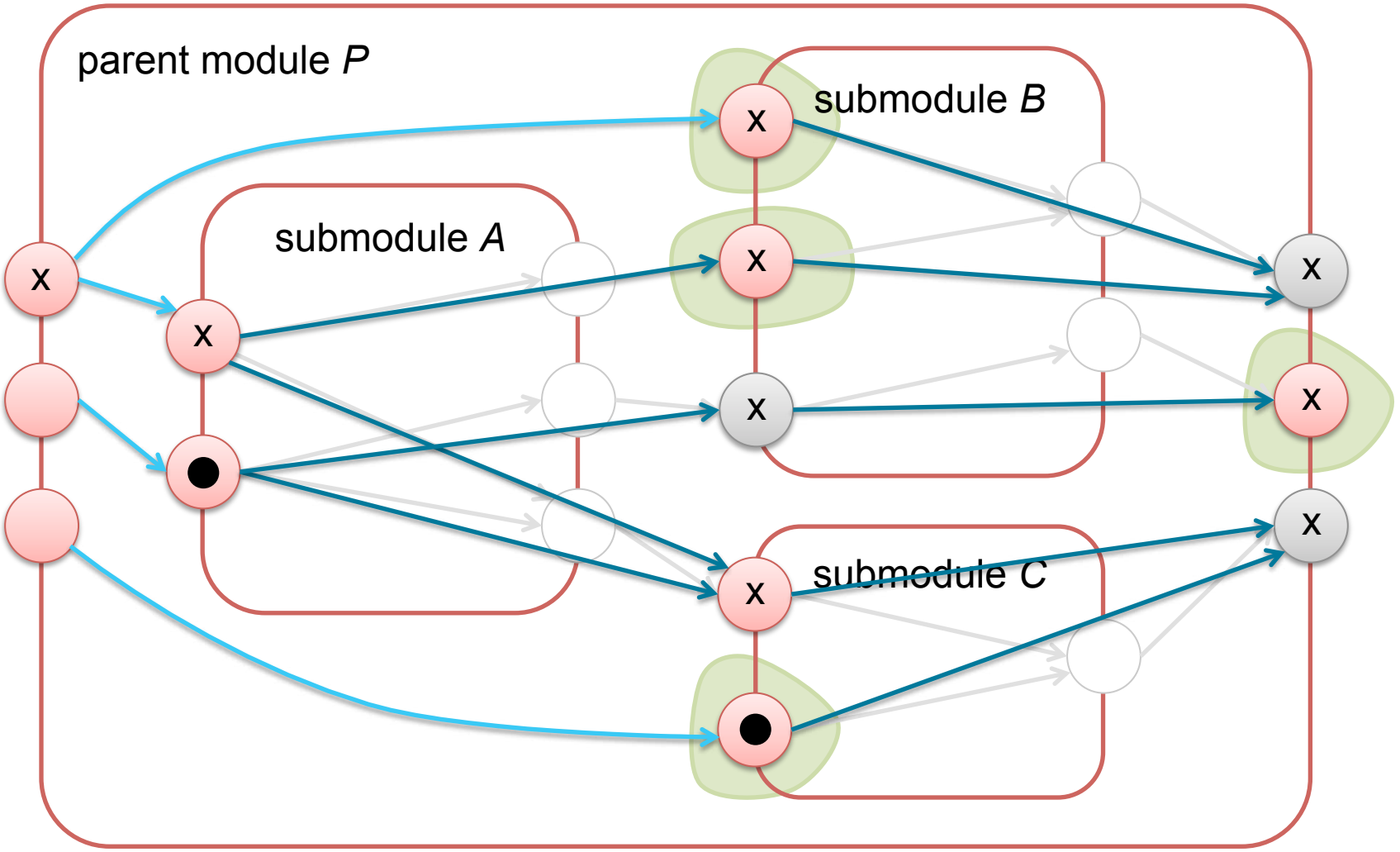
Restarting Workflows, Dependency Graph



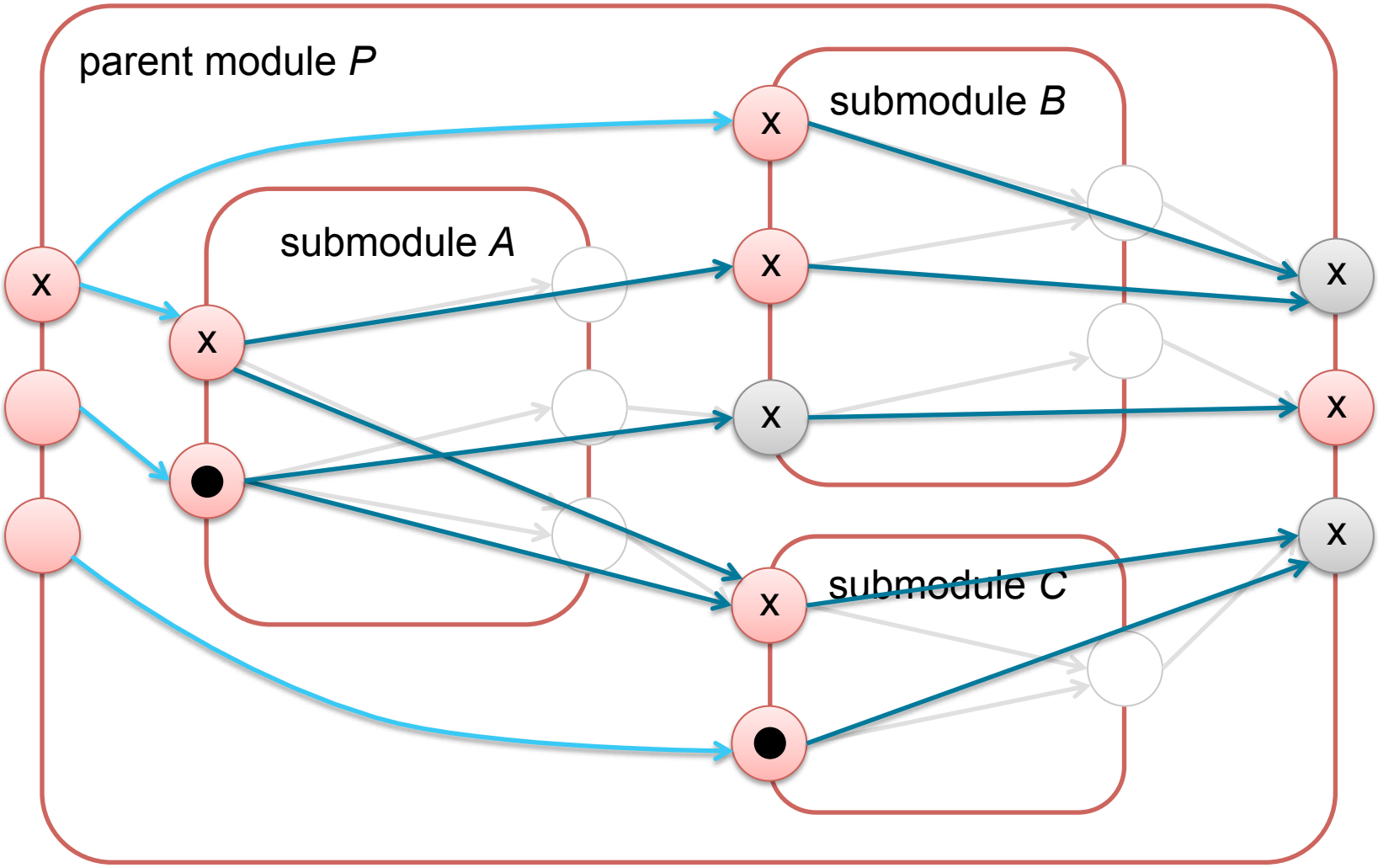
Restarting Workflows, Dependency Graph



Restarting Workflows, Dependency Graph



Restarting Workflows, Dependency Graph



The Staging-Area Abstraction

Support for arbitrary back ends

From in-memory data structures to file systems and databases

The Staging-Area Interface

High-Level

- Methods every interpreter needs (whether it works on simple, composite, or any other module)
- Superficially similar to key-value store, but:

```
public interface StagingArea {
    Future<RuntimeExecutionTrace> delete(RuntimeExecutionTrace prefix);
    Future<RuntimeExecutionTrace> copy(RuntimeExecutionTrace source,
        RuntimeExecutionTrace target);
    Future<RuntimeExecutionTrace> putObject(RuntimeExecutionTrace target,
        Object object);
    // ...
```

  - Keys are execution traces that capture call stack plus the port name and possibly array indices
  - Handles object marshaling if necessary
- Could be backed by in-memory Java data structures, a file system, a database, etc.

Object Marshaling

Requirements

- Choice of marshaler should be kept as metadata only (loose coupling)
- CloudKeeper should perform dependency resolution (package management) for marshalers
  - Little/no user configuration at runtime
- Possibility for user to override choice of marshaler (per execution)
- Marshalers must support third-party classes
- Executor component should not need to perform class loading
  - Notion of array indices built into staging-area abstraction

No class (un-)loading worry when running CloudKeeper as a service!

Staging Areas Provide Marshaling Contexts

User-Defined Object Marshaling

```
public interface Marshaler<T> {
    void put(T object, MarshalContext context) throws IOException;
    T get(UnmarshalContext context) throws IOException;
    // ...
}
```

- `class S extends Marshaler<T>` can handle type `U` if `T :> U`
- Collection of key/stream pairs (key is index, identifier, or empty)

```
public interface MarshalContext {
    OutputStream newOutputStream(Key key) throws IOException;
    void putByteSequence(ByteSequence byteSequence, Key key) throws IOException;
    void writeObject(Object object, Key key) throws IOException;
}
```

Marshal Context Provided by Staging Area

- `writeObject()` chooses `Marshaler` implementation or handles object directly, based on `object.getClass()`

Defaulting to Java Serialization

CloudKeeper Provides Default Serialization

- Fallback for all Java `Serializable` objects (includes a lot)

```
@SerializationPlugin("Serialize objects that implement the Serializable interface.")
public final class SerializableMarshaler implements Marshaler<Serializable> {
    @Override
    public void put(Serializable object, SerializationContext context)
            throws IOException {
        try (ObjectOutputStream objectOutputStream
                = new ObjectOutputStream(context.newOutputStream(Token.empty()))) {
            objectOutputStream.writeObject(object);
        }
    }
    // ...
```

- For boxed types (Integer, Long, …), simple as-string marshaler has higher precedence by default

Recursive Serialization of Collections

```
public final class CollectionSerialization implements Serialization<Collection<?>> {
    private static final Identifier SIZE = Identifier.identifier("size");

    @Override
    public void put(Collection<?> collection, MarshalContext context)
            throws IOException {
        int count = 0;
        context.writeObject(collection.size(), SIZE);
        for (Object object: collection) {
            context.writeObject(object, Index.index(count));
            ++count;
        }
    }

    @Override
    public Collection<?> get(UnmarshalContext context) throws IOException {
        int size = (int) context.readObject(SIZE);
        List<Object> list = new ArrayList<>(size);
        for (int i = 0; i < size; ++i) {
            list.add(context.readObject(Index.index(i)));
        }
        return list;
    }
}
```

CloudKeeper Customization

Metadata via Annotations

Type declarations

All Metadata Kept as Annotations

Example: User-Defined Annotations

- Define annotation for resource requirements

```
@AnnotationTypePlugin("Memory requirement in GB.")
public @interface Memory {
    int value();
}
```

- Retrieve annotation in customized simple-module executor

```
@Nullable Memory requirements = trace.getAnnotation(Memory.class)
```

- Apply to module, either on the declaration or on an instance

```
@Memory(12)
AvgLineLengthModule avgLineLengthModule = child(AvgLineLengthModule.class)
    .text().from(reads());
```

Using Annotations for Customization

Annotation Inheritance

- More complicated than in Java
  - Module > Module declaration
  - Type declaration > Super-class type declaration
  - Port > Port in super-module declaration (later)

Override Annotations Per Execution

- for particular “execution trace”
- for particular element of declaration
- for one of the previous when conforming to a pattern (regular expression)

```
execution.setOverrides(Arrays.asList(
    new MutableExecutionTraceOverride()
        .setTrace("/avgLineLengthModule")
        .setAnnotations(Arrays.asList(
            new MutableAnnotation()
                .setDeclarationName(Memory.class.getName())
                .setElements(Arrays.asList(
                    new MutableAnnotationElement()
                        .setName("value")
                        .setValue(12)
                ))
        ))
));
```

Declaration: CloudKeeper Types

Declaration

- Type declaration = Class or interface with `@TypePlugin` annotation
- Cannot be inner class (that is, nested class without **`static`** keyword)
- Real example: **`public interface`** `ByteSequence`
- System repository has declarations for standard types (boxed types, `String`, `Serializable`, and a few others)

Metadata

- Default serialization to use when not overridden
- Also `Collection`, despite its special semantics, uses serialization infrastructure

Declaration of Existing Types

Problem

- Cannot add annotations to existing classes/interfaces (`Object`, `Collection`, ...)

Solution

- Mixins: Use annotations on class A for class B
- Mapping: Remove prefix `cloudkeeper.mixin.` from qualified name
- Example:

```
package cloudkeeper.mixin.java.lang;

import com.svbio.cloudkeeper.dsl.TypePlugin;

@TypePlugin(description = "Root type.")
public final class Object { }
```