Analysis of Software Artifacts

Mini-Eclipse:
Checking a Framework with Plural

Jonathan Aldrich

Mini-Eclipse

- Simple framework
  - Demonstration of Plural verification ideas
- Models a few simple features of Eclipse
  - Plugin lifecycle
  - Menus
  - Notification messages

The Plugin Lifecycle

```
@States({"created", "started", "stopped"})
public interface Plugin {
    @Full(requires = "created", ensures = "started")
    public void start(                    FrameworkFacade fwk);
    @Full(requires = "started", ensures = "stopped")
    public void stop(                     FrameworkFacade fwk);
}
```

Protocol captures order of lifecycle methods

Need full permission because the plugin will be changing its state

In a real example many other methods might be called in the started state (where most of the action happens)

The Plugin Lifecycle

```
@States({"created", "started", "stopped"})
public interface Plugin {
    @Full(requires = "created", ensures = "started")

    public void start(@Full("setup") FrameworkFacade fwk);

    @Full(requires = "started", ensures = "stopped")

    public void stop(@Full("teardown") FrameworkFacade fwk);
}
```

The start method has to be able to affect the framework, so we give it a @Full permission (@Share would also work).

Different framework methods can be called from start() vs. stop() vs. other methods. We use the setup and teardown framework states to encode this.

The Plugin Lifecycle

```
@States({"created", "started", "stopped"})
public interface Plugin {
    @Full(requires = "created", ensures = "started")

    public void start(@Full("setup") FrameworkFacade fwk);

    @Full(requires = "started", ensures = "stopped")

    public void stop(@Full("teardown") FrameworkFacade fwk);
}
```

In this design the Plugin does NOT get to keep a reference to the framework—to allow it to keep a reference, we would have to set returned = false on our @Full annotation. We want to avoid the Plugin keeping a reference because the framework will have to change state, and that shouldn't interfere with the plugin.

The Plugin Lifecycle

```
@States({"created", "started", "stopped"})
public interface Plugin {
    @Full(requires = "created", ensures = "started")
    @Perm(requires="#0 != null")
    public void start(@Full("setup") FrameworkFacade fwk);

    @Full(requires = "started", ensures = "stopped")
    @Perm(requires="#0 != null")
    public void stop(@Full("teardown") FrameworkFacade fwk);
}
```

We'll also ensure that the framework parameter is non-null. Note that in Plural we have to refer to a parameter by its position. That's because parameter names in Java aren't stored in the bytecode.

The Framework Façade

```
@States(dim="PROTOCOL", value= {"setup", "running", "teardown"})
public interface FrameworkFacade {
    @Full("setup")

    public void registerMenu(String itemName, @Share Action
        menuAction);

    @Full("running")

    public void notifyWithPopup(String message);
}
```

Menus can only be set up in the “setup” phase of the Framework (when Plugin.start() is called)

We make the menu action @Share so the same action object can be reused in menus and toolbar buttons. It can’t be @Pure or @Imm because the menu action is going to change system state.

The Framework Façade

```
@States(dim="PROTOCOL", value= {"setup", "running", "teardown"})
public interface FrameworkFacade {
    @Full("setup")
    @Perm(requires="#0 != null * #1 != null")
    public void registerMenu(String itemName, @Share Action
        menuAction);

    @Full("running")
    @Perm(requires="#0 != null")
    public void notifyWithPopup(String message);
}
```

We'll do more null checking.

Actions

```
public interface Action {

@Share
@Perm(requires="#0 != null")
public void run( @Full("running") FrameworkFacade fwk);
}
```

When a menu action is invoked the Framework is in the running state.

Verifying a Plugin

```
public class SimplePlugin implements Plugin {
    @Perm(ensures="unique(this!fr) in created")
    public SimplePlugin() {}
```

The constructor returns a unique object in the created state—the state from which the framework will start() the Plugin.

Note that this is an ensures permission, a postcondition and not a precondition. We have the permission to the object at the end of the constructor but it is not passed in to the method. Instead the system created the object for us and we initialized it.

When we are setting up an object in the constructor, or changing its state in a method (directly, not by calling other methods), Plural requires a *frame permission* denoted by this!fr (rather than this). The reasons are technical and have to do with inheritance.

Verifying a Plugin

```
public class SimplePlugin implements Plugin {
    @Perm(ensures="unique(this!fr) in created")
    public SimplePlugin() {}

    @Override
    @Full(requires = "created", ensures = "started",
        fieldAccess=true)
    public void start(@Full("setup") FrameworkFacade fwk) {
        fwk.registerMenu("A Menu Item", new SimpleAction());
    }
```

The start method is annotated exactly like the interface—except here we are going to change the state of the plugin, so we need a frame permission. When using @Full instead of @Perm, we set the fieldAccess attribute to **true**.

Initializing the Framework

```
public class Main {
    @Perm(requires="#0 != null")
    public static void run(
                @Full(requires="created", returned=false) Plugin p) {
        new FrameworkImpl(p).run();
    }
}
```

Start the framework by calling a static method (we could have also used the Singleton pattern)

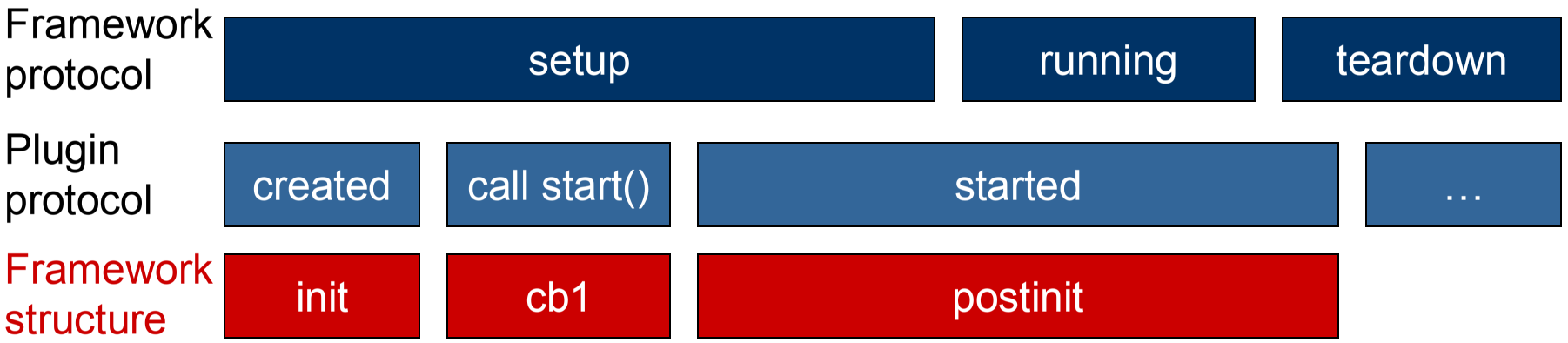
We pass the plugin to the run method. Note that the permission is not returned to the client. Eclipse does this slightly differently—it looks in a configuration file then uses dynamic classloading and reflection to create the plugin.

Verifying the Framework

- Challenge: state invariant for the plugin
  - Does not match framework state exactly

Verifying the Framework

- Challenge: state invariant for the plugin
  - Does not match framework state exactly
  - So we add a new STRUCTURE state dimension for the framework that matches the plugin state transitions



Framework States and Invariants

```
@States(dim="PROTOCOL", value= {"setup", "running", "teardown"})
@States(dim="STRUCTURE", value= {"init", "cb1", "postinit"})
@ClassStates({

      @State(name="init", inv="full(myPlugin) in created"),
      @State(name="postinit", inv="full(myPlugin) in started"),
})
public class FrameworkImpl implements FrameworkFacade {
      private Plugin myPlugin;
```

When the framework is in the init state, myPlugin is in the created state

When the framework calls myPlugin.start(), it passes the full permission to start(). Now the framework doesn't have it any more; the framework is in the cb1 state, where there is no permission to myPlugin.

When the framework is in the postinit state, after start() returns, myPlugin is in the started state

Framework States and Invariants

```
@States(dim="PROTOCOL", value= {"setup", "running", "teardown"})
@States(dim="STRUCTURE", value= {"init", "cb1", "postinit"})
@ClassStates({
    @State(name="PROTOCOL", inv="share(myAction)"),
    @State(name="init", inv="full(myPlugin) in created"),
    @State(name="postinit", inv="full(myPlugin) in started"),
})
public class FrameworkImpl implements FrameworkFacade {
    private Plugin myPlugin;
    private Action myAction;
```

myAction can be part of the main PROTOCOL dimension; we just have a share permission to it.

The Framework Constructor

```
@States(dim="PROTOCOL", value= {"setup", "running", "teardown"})
@States(dim="STRUCTURE", value= {"init", "cb1", "postinit"})
public class FrameworkImpl implements FrameworkFacade {
    private Plugin myPlugin;
    private Action myAction;
    @Perm(requires="full(#0) in created * #0 != null",
          ensures="full(this!fr) in setup, init")
    public FrameworkImpl(Plugin p) {
        myPlugin = p;
        myAction = null;
```

The constructor requires a full permission to a non-null Plugin in the created state. We must specify a state for each dimension in the postcondition.

Note that Plural requires us to explicitly initialize myAction to **null** in the constructor (the tool does not yet recognize default initialization to **null**).

Advanced Framework Verification

(the homework does not require you succeed at this with Plural)

```
@Perm(requires="full(this) in setup,init * this != null")
public void run() {
    doSetup(this);
    … }
```

We get a full permission to the framework in the setup and init states (for the PROTOCOL and STRUCTURE dimensions, respectively)

```
@Full(value="STRUCTURE", requires="init", ensures="postinit",
        fieldAccess = true)
@Perm(requires="#0 != null")
public void doSetup(@Full("setup") FrameworkFacade fwk) {
    myPlugin.start(fwk);
}
```

Why call doSetup?

We need to access the myPlugin field, so we need a frame permission (fieldAccess) to the STRUCTURE dimension.

But we need to pass a non-frame permission to the PROTOCOL dimension to start(). Plural splits up the permission for the call.

Advanced Framework Verification

(the homework does not require you succeed at this with Plural)

```
@Perm(requires="full(this) in setup,init * this != null")
public void run() {
    doSetup(this);
    switchToRunning();
    … }
```

We need to switch to the "running" state. This doesn't actually change our state, but Plural requires we call a method to do it. Thus the switchToRunning() dummy method.

```
@Full(value="PROTOCOL", requires="setup", ensures="running",
    fieldAccess=true)
protected void switchToRunning() {
}
```

Note that here we need a frame permission (fieldAccess) because we are changing the state, even though we aren't actually modifying any fields. This is a current Plural limitation.

Note that we only get a permission to the PROTOCOL dimension. This lets plural know we aren't changing the STRUCTURE dimension in any way.

Advanced Framework Verification

(the homework does not require you succeed at this with Plural)

```
public void run() {
        … Action action = getAction();
        doRunning(this, action);
        … }


@Full(value = "running", fieldAccess = true)
@ResultShare()
protected Action getAction() {
        return myAction;
}


@Perm(requires="#0 != null")
public void doRunning(    @Full("running") FrameworkFacade fwk,
                          @Share Action action) {
        action.run(fwk);
}
```

Why can't we do getAction() inside doRunning()?

Because doRunning() needs a ***non-frame*** full permission in the PROTOCOL dimension, which is where myAction is stored, to pass to action.run(). Since we don't have a frame permission, we can't access myAction in doRunning.

Fortunately myAction is @Share so we just call getAction first and make a copy of the reference.