I-X Process Panels – Developer Guide

Jeff Dalton
Artificial Intelligence Applications Institute
Centre for Intelligent Systems and their Applications
School of Informatics
The University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN, UK

Web: http://i-x.info
E-mail: query@i-x.info

Version 2.3 – 3rd October 2002

1 Providing a Custom World State Viewer

This document is about I-X version 2.3. Many of the details given here will change, and the change may be soon; but the changes required in a viewer written for 2.3 should not be difficult to make.

1.1 What does the viewer do?

A world-state viewer is responsible for displaying the panel's model of the current state of the world. The state is a set of pattern-value pairs. For example, using one possible textual notation:

```
colour block-1 = red
size block-1 = small
position block-2 = (10 30)
position robot = (100 10)
```

To the left of the "=" is the "pattern", to the right, the "value". This is mere terminology.

The default value is "true". Anything not specified in the state is unknown. There is no closed-world assumption that says everything not specified is false.

The viewer is told about changes in the state. It isn't (in 2.2) ever given a whole state to display, although the first change it sees will be the entire state at that time. This means that most world-state viewers will have to maintain their own record of the whole state.

The viewer must also be able to "reset" - to clear any record or display of what the state is.

1.2 How do I change the viewer used by I-P^2?

Define a subclass of ix.ip2.Ip2 that redefines the following method:

```
    /**
     * Called to create the state viewer.  This method can be
     * redefined in subclass that want to instantiate a different
     * viewer class.
     */
    protected StateViewer makeStateViewer() {
        return new StateViewTable(this);
    }
```

Instantiate whatever viewer class you desire instead of StateViewTable.

1.3 How do I define a new viewer class?

The class ix.ip2.StateViewTable may be used as an example. It corresponds to file java/ix/ip2/StateViewTable.java in the I-X distribution.

The class must:

- implement the StateViewer interface;
- be a subclass of java.awt.Component;
- have a 1-argument constructor that takes an instance of Ip2 as a parameter. (In the constructor declaration a superclass of Ip2, such as IXAgent, may be specified instead.)

In I-X 2.2, the StateViewer interface extends the ProcessStatusListener interface. This is may change. Until it does, however, you must define some methods that will never actually be called. Here are the ones that WILL be called:

```
    public void reset();

    public void stateChange(ProcessStatusEvent e, Map delta);
```

The ones that will not be called are:

```
    public void statusUpdate(ProcessStatusEvent e);

    public void newBindings(ProcessStatusEvent e, Map bindings);
```

Here is an outline viewer based on the above:

```
package ix.ip2;         // or whatever package you desire

import java.util.*;
import javax.swing.*;
import ix.icore.process.event.*;
import ix.util.*;
import ix.util.lisp.*;

public class MyStateViewer extends JPanel implements StateViewer {

    /**
     * Constructs a viewer for the indicated agent.
     */
    public MyStateViewer(IXAgent ip2) {
        super();
    }

    /**
     * Sets the viewer back to something approximating its initial state.
     */
    public void reset() {
        // Clear state data
    }

    /*
     * Methods in ProcessStatusListener interface
     */

    /** Ignored by this viewer. */
    public void statusUpdate(ProcessStatusEvent event) { }

    /** Ignored by this viewer. */
    public void newBindings(ProcessStatusEvent event, Map bindings) { }

    public void stateChange(ProcessStatusEvent event, Map delta) {
        for (Iterator i = delta.entrySet().iterator(); i.hasNext();) {
            Map.Entry e = (Map.Entry)i.next();
            LList pattern = (LList)e.getKey();
            Object value = e.getValue();
            recordNewValue(pattern, value);
        }
    }

    protected void recordNewValue(LList pattern, Object value) {
        Debug.noteln("State viewer sees " + pattern + " = " + value);
        // ...
    }

}
```

1.4 How are patterns and values represented?

As you can see in the code above, a pattern is an LList and a value can be any Object. An LList is a class of singly-linked, "Lisp-like" lists. In the viewer, you can usually ignore the fact that that particular class is used and use the java.util interface List in declarations instead. But when constructing a pattern, or a
pattern-element that is a List, an LList must be used.

Although values and pattern-elements can be any Object, they will in practice always come from a restricted set of classes:

- Symbols - instances of ix.util.lisp.Symbol.
- Strings - instances of the class String.
- Numbers - instances of classes Long, Double, etc.
- Variables - instances of ix.util.lisp.ItemVar.
- Lists - instances of ix.util.lisp.LList

Variables will not appear in any pattern or value in the world state.

To create a Symbol, call Symbol.intern(String name). If a Symbol of that name already exists, it is returned; otherwise a new Symbol is constructed. This ensures that there is only one Symbol per name and hence that they can be compared with ==.

For example:

```
 static final Symbol S_REQUEST = Symbol.intern("request");
 ...
   if (pattern.get(0) == S_REQUEST) ...
```

An ItemVar is obtained by using a name that begins with "?".

There are a number of ways to construct LLists. Here are some of the most common:

```
Lisp.NIL                    // the empty LList
Lisp.cons(Object, LList)    // for recursive construction
Lisp.list(Object)
Lisp.list(Object, Object)
Lisp.list(Object, Object, Object)
Lisp.list(Object, Object, Object, Object)
... up to 5 arguments ...

LList.newLList(Collection)

N.B. new LList(Collection) does NOT work.

(LList)Lisp.readFromString(String),
   where the String is a list in Lisp-like syntax.
```

The Lisp class is ix.util.lisp.Lisp.

You can also match against patterns (treated as data) by constructing a pattern that contains ItemVars. For example:

```
import ix.util.lisp.*;
import ix.util.match.*;

static final Symbol
   Q_AGENT = Symbol.intern("?agent"),
   Q_CONTENTS = Symbol.intern("?contents");

LList syntax = Lisp.readFromString("(send_to ?agent ?contents)");

LList data = ...;

MatchEnv env = SimpleMatcher.match(syntax, data);

...
if (env != null) {
   Symbol agentName = (Symbol)env.get(Q_AGENT);
   Object messageContents = env.get(Q_CONTENTS);
   ...
}
```

There are more sophisticated ways to do this kind of thing, but they are "beyond the scope of this document".

2 Providing a Communications Strategy

2.1 Overview

I-X Process Panels can also be used with any of a number of “Communications Strategies”. "Strategy" is too grand, but "method" could be confusing in Java. Example strategies are provided for the DARPA CoABS Grid (“grid”), the University of West Florida Institute of Human and Machine Cognition (UWF/IHMC) KAoS (“kaos”), the Jabber (www.jabber.org) XML framework (“jabber”), and the UK EPSRC-sponsored Advanced Knowledge Technologies AKT Bus (“akt”). Also provided is an adaptor for a simple direct link between panels possibly supported by a simple name server (referred to as the “simple” or “xml” communications strategy).

Writing a suitable Communications Strategy can provide other message transport routes. The code should be placed in the appropriately named location in the comms directory, and a suitable comms-setvar.bat script provided within the relevant directory. The I-X system architecture potentially supports the dynamic addition of communications strategies on-the-fly.

From the point of view of the rest of the system, a communication strategy is responsible only for sending and receiving messages, and it's ordinary Java objects that are sent and received. The rest of the system knows nothing about how an object might be encoded when it's sent, and it doesn't have to construct special message objects in order to send something.

However, there are some restrictions. A comm strategy doesn't have to be able to handle every kind of Object. For example, the "simple" strategy requires that the objects be Serializable, and the "xml" strategy requires that they be defined in ways that can be handled by the I-X XML-translation code. (They have to have get- and set-methods for the fields that need to be encoded, for example.)

However, all strategies have to be able to handle the objects that we're actually going to send: instances of Issue, Activity, etc. This list will get longer, so special-case code should not be used. However, these objects will all be ones that the XML translator can handle, and that means there's support for extracting the relevant fields, etc, which can be used independently of the specifically XML-related code. They will also all be serializable.

Moreover, if a new strategy wants to use an encoding that we'd like to support (an example might be the Java 1.4 XML serialization for beans), the class definitions of the objects we want to send will be adjusted as required, or else some kind of translation support would be developed.

(A strategy that wants to use an XML encoding should use the usual I-X translation code unless there's a good reason not to.)

A strategy normally supports only one way of doing things and so can be used only with agents that are using a compatible approach. A kind of "meta strategy" that picks the required strategy for each case could be created, but so far this has not been necessary.

2.2 Some Interfaces

Many objects that we send will implement the Sendable interface, which presently contains

```
  public Name getSenderId();
  public void setSenderId(Name id);
  public Object clone() throws CloneNotSupportedException;
```

It is not required that they implement that interface, but if they do, the comm strategy might find the sender-id useful in error messages or for other purposes.

The main strategy-related interfaces have "IPC." at the front of their name, because they are all defined inside the ix.util.IPC class.

The strategy itself must implement the IPC.CommunicatonStrategy interface which contains:

```
   public void sendObject(Object destination, Object contents);
   public void setupServer(Object destination,
                     IPC.MessageListener listener);
```

When a new message arrives, it should be passed to a message-listener object that implements the IPC.MessageListener interface:

```
   public void messageReceived(IPC.InputMessage message);
```

IPC.InputMessage is another interface, containing:

```
   public Object getContents();
```

So there are special message objects on the *receiving* side. It is also possible for them to be created *inside* the sendObject method. This allows the communication strategy to include any information it wants to "wrap" around the message contents when sending and, on the receiving end, to give this information to the message-listener in case it is useful for debugging output or some other purpose. Of course, further "wrapping" may also be used, but it would be invisible to the message-listener as well as to the code that calls sendObject.

A minimal implementation of IPC.InputMessage, IPC.BasicInputMessage, is provided. It has a 1-argument constructor that takes the contents object as a parameter.

2.3 Sending

```
   public void sendObject(Object destination, Object contents);
```

The sendObject method is responsible for sending the contents to the destination. So in that case, the destination identifies the remote agent we're sending to. Code that calls this method will expect to be able to catch an exception if anything goes wrong. SendObject should not return until the send has (so far as it knows) completed. If for some reason a strategy would need to do some of the sending in another thread, and it's impractical to wait until that finishes, we will have to decide on a case-by-case basis what should be done. It is possible that a new mechanism for reporting such problems would be created (and could then be used by other strategies).

In practice, the destinations will all be Strings, but it's best for the comm strategy not to depend on that.

The sendObject method will often be called from the AWT / Swing event-dispatching thread, but that should not be assumed. (See the JDK documentation for SwingUtilities.invokeAndWait(Runnable) and SwingUtilities.invokeLater(Runnable) to see how to deal with such situations.)

However, within the comm strategy, sending should not normally try to do anything with the GUI; it should just report problems by throwing exceptions.

Note that these destination objects are meant to be meaningful at the user level, and the same ones will be used with all comm. strategies. They are not whatever low-level name or address the comm strategy ultimately uses. Some comm strategies use an agent-communication package that can use "destinations" directly as agent names (at least the destination objects we actually use). In other cases, some way of translating between I-X "destinations" and lower-level entities must be provided.

2.4 Receiving

```
  public void setupServer(Object destination,
                   IPC.MessageListener listener);
```

The setupServer method is responsible for doing whatever is required to allow this agent to receive messages. In this case, the "destination" parameter is the object that should be used to refer to this agent when remote agents want to send to it. It is not strictly necessary to communicate this destination object to the remote agents, just so long as they are somehow able to send to us, but it will typically be used in some sort of registration process for this agent and thus become known to the remote ones.

The setupServer method is called exactly once. It is called from the main thread, during the initialisation of the I-X agent, and before the agent's GUI has been created.

In any case, setupServer must return once it thinks it has things set up, having created any additional threads needed to handle the actual receiving of messages.

Those threads are meant to run independently of the rest of the system and to report problems to the user via the SwingUtilities.invokeAndWait method.

The IPC.MessageListener used by I-X agents has a synchronized messageReceived method in case multiple receiving threads want to call it. It also handles the transition to the thread that is running the main body of the agent.

When an existing agent-communication package / system is used as the basis of a communication strategy, it may not be visible in the source code written to use that package that any threads are being created; but they must be for the rest of the I-X agent to be able to run independently. You may therefore need to be aware of this when writing the strategy.

If the package does not create any message-receiving threads, you will have to do it yourself.

2.5 Using/Testing a Strategy

The strategy is specified when an I-X agent is run. It can be specified on the command line or in a .props file. The details of this will depend on what scripts you are using to run I-X agents, but the aim is to give a strategy name as the value of the "ipc" parameter. For example, using a script that approximates the normal "java" command (but with the class-path set up for I-X):

```
  scrips/unix/ix-java ix.ip2.Ip2 -ipc=simple
```

A strategy may require or allow additional parameters, for example:

```
  scrips/unix/ix-java ix.ip2.Ip2 -ipc=simple -run-name-server
```

These should tyically be examined in the setupServer method. For information on how to get parameter values, see the javadoc for the class ix.util.Parameters. It is not necessary that any additional parameters be supported, however. "run-name-server" is specific to the "xml" and "simple" strategies.

The "-" in "-ipc" is part of the syntax for command-line arguments, not part of the parameter name, which is just "ipc". So in a .props file, the "-" would not appear.

The above description is not meant to be enough unless you already know how to run I-X agents and to specify such "parameters", but it should give you the basic idea, namely that there is a parameter called "ipc", specified when the agent is started, that has a value that is the name of the communication strategy to be used.

Now, this strategy-name can be a class name, or it can be an abbreviation. "simple", "xml", and "grid" are examples of abbreviations. If the class name (and package) of your new communication strategy takes a certain form, an abbreviation is automatically defined. Otherwise, the full, package-qualified, class name will have to be used. For an abbreviation, name, the full class name is ix.name.NameCommunicationStrategy. For instance the abbreviation "grid" corresponds to

```
  ix.grid.GridCommunicationStrategy
```

You might wonder how you can get something into the "ix" package without making it part of the main I-X code. The answer is that any "ix" directory can be used, so long as it is on the class-path.

Note that when an I-X agent's "Messenger" is used to send to "me", the message does not go through the communication strategy at all; but if you send to the agent's own "destination", it does. All communication strategies should handle an agent sending to itself in that way.

The destination an agent uses for itself (and thus for other agents to send to it) can be specified via the "symbol-name" parameter. For example,

```
scrips/unix/ix-java ix.ip2.Ip2 -symbol-name=jack -ipc=xml -run-name-server
```

Finally, note that some debugging messages that appear with all communication strategies print XML representations of objecs using the standard I-X encoding. Don't be misled by this if you are using a different XML encoding or are not using XML at all.