Soar Technology
Thinking *inside* the box.

Soar Integration Lessons Learned

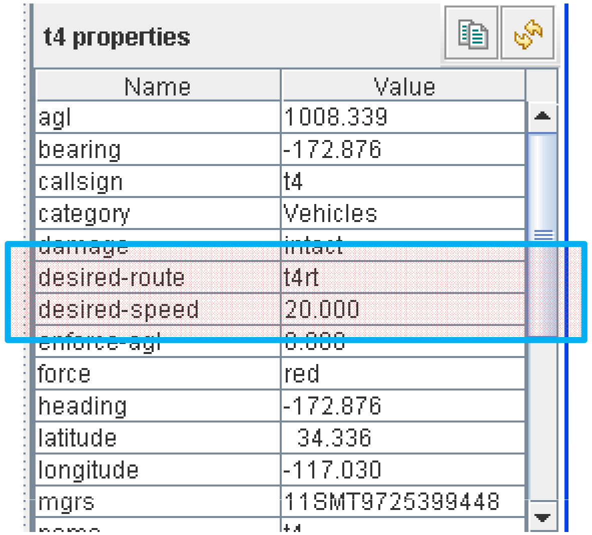
2008 Soar Workshop
Dave Ray
ray@soartech.com

Overview

- Make debugging easier
- Apply basic software engineering to Soar systems
- Test your code
- Get along with SML

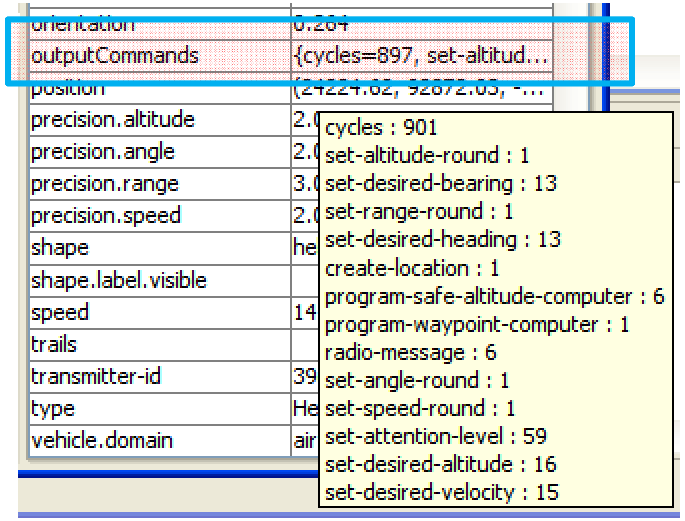
Make it easy to tell what's going on

- Give agents and SSI code a place to put useful debugging information
- Poor man's VISTA (SoarTech's, not Microsoft's)
- Here, SSI code in Sim Jr can set arbitrary named properties
- Easy to reference at runtime
- Not a replacement for a good log



t4 properties

Name	Value
agl	1008.339
bearing	-172.876
callsign	t4
category	Vehicles
damage	intact
desired-route	t4rt
desired-speed	20.000
enforce-agl	0.000
force	red
heading	-172.876
latitude	34.336
longitude	-117.030
mgrs	11SMT9725399448
name	t4

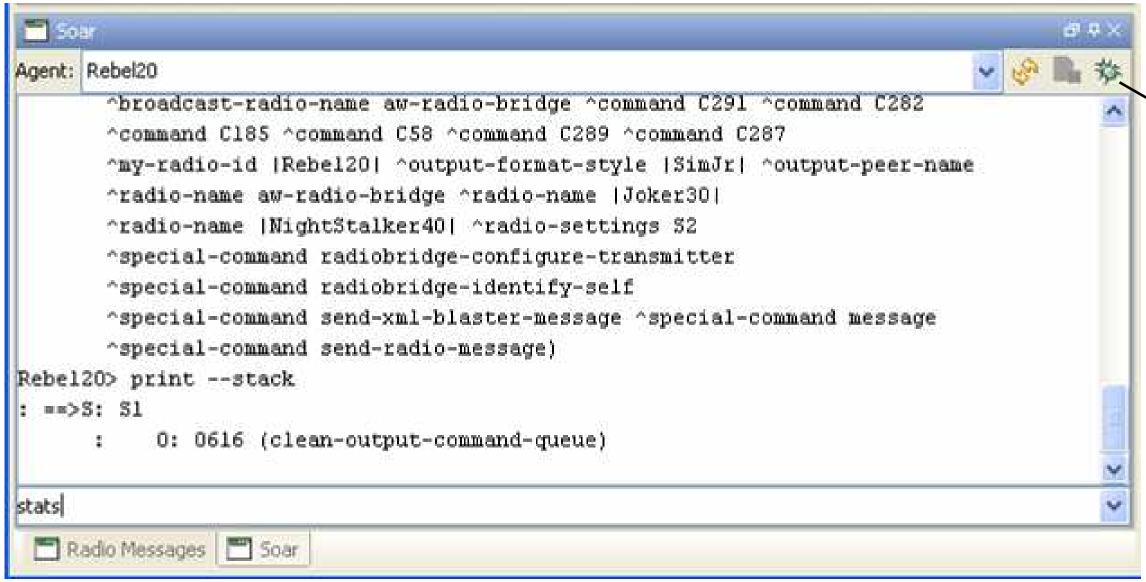


orientation	0.264
outputCommands	{cycles=897, set-altitud...
position	{24224.62, 92872.03, -...
precision.altitude	2.0
precision.angle	2.0
precision.range	3.0
precision.speed	2.0
shape	he
shape.label.visible	
speed	14
trails	
transmitter-id	39
type	He
vehicle.domain	air

cycles : 901
set-altitude-round : 1
set-desired-bearing : 13
set-range-round : 1
set-desired-heading : 13
create-location : 1
program-safe-altitude-computer : 6
program-waypoint-computer : 1
radio-message : 6
set-angle-round : 1
set-speed-round : 1
set-attention-level : 59
set-desired-altitude : 16
set-desired-velocity : 15

Make it easy to tell what's going on

- Startup and run is slower with TSI or Java debugger
- Create a simple command window for inspecting agent state
- No print output so no overhead
- Window shown is ~100 lines of Java code in Sim Jr

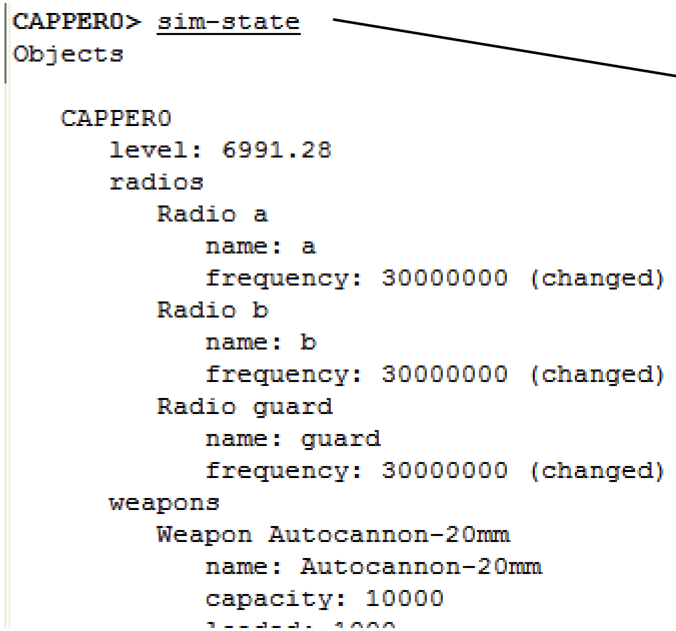


```
        ^broadcast-radio-name aw-radio-bridge ^command C291 ^command C282
        ^command C185 ^command C58 ^command C289 ^command C287
        ^my-radio-id |Rebel20| ^output-format-style |SimJr| ^output-peer-name
        ^radio-name aw-radio-bridge ^radio-name |Joker30|
        ^radio-name |NightStalker40| ^radio-settings S2
        ^special-command radiobridge-configure-transmitter
        ^special-command radiobridge-identify-self
        ^special-command send-xml-blaster-message ^special-command message
        ^special-command send-radio-message)
Rebel20> print --stack
: ==>S: S1
      :    O: O616 (clean-output-command-queue)
```

Open a real Soar debugger when you need it

Make it easy to tell what's going on

- Use SML filters to create custom debugging commands
- Callable from Java Soar debugger and Soar files

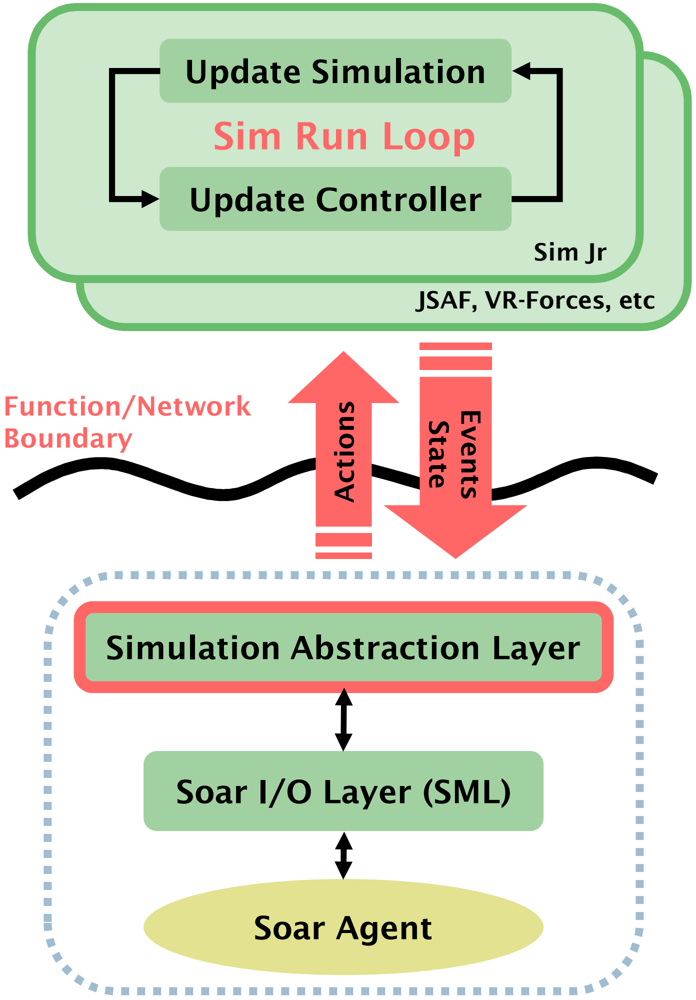


```
CAPPERO> sim-state
Objects

   CAPPERO
      level: 6991.28
      radios
         Radio a
            name: a
            frequency: 30000000 (changed)
         Radio b
            name: b
            frequency: 30000000 (changed)
         Radio guard
            name: guard
            frequency: 30000000 (changed)
      weapons
         Weapon Autocannon-20mm
            name: Autocannon-20mm
            capacity: 10000
            loaded: 1000
```

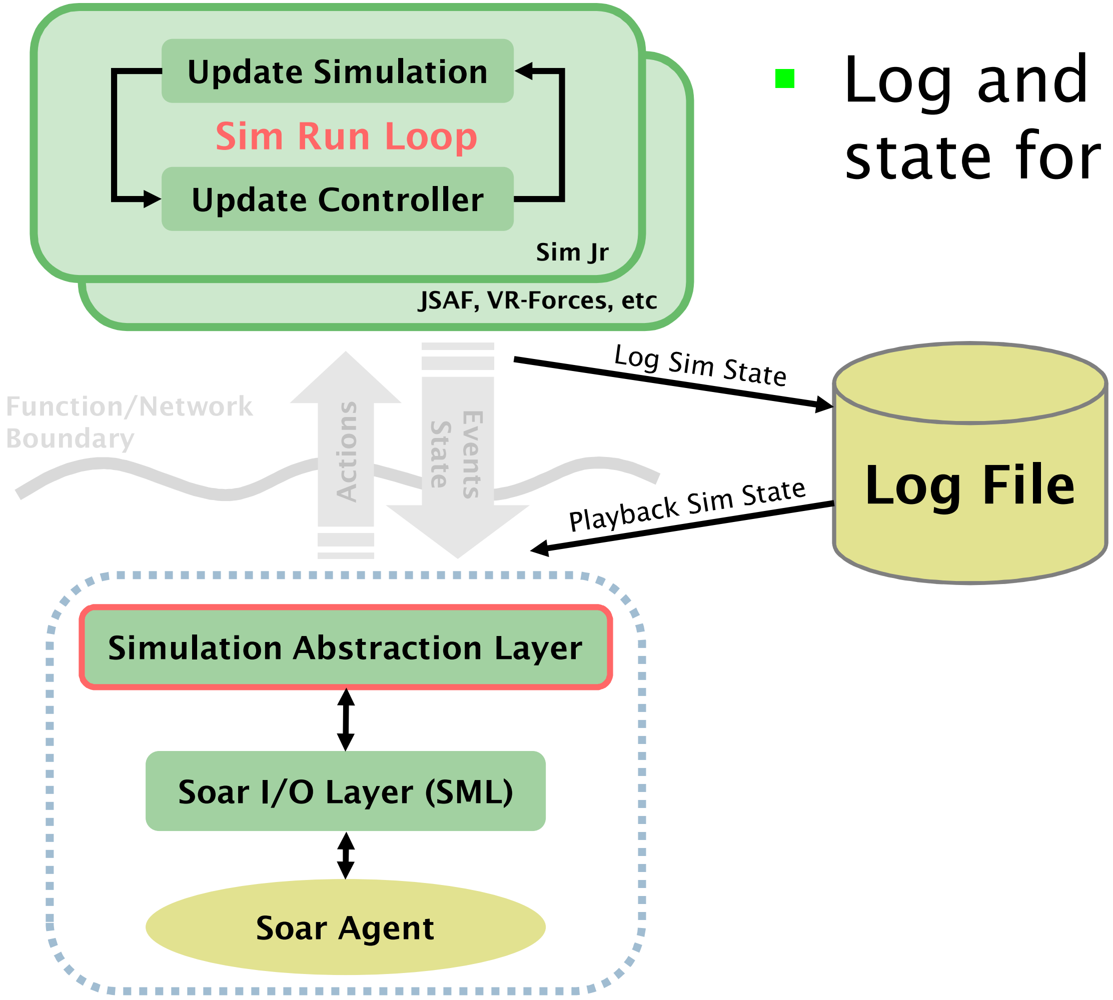
sim-state command prints out the current state of our simulation abstraction layer

Create a Simulation Abstraction Layer



- Create an abstraction layer for your simulation/environment
- Code Soar I/O to abstraction rather than simulation API
- Easier to move to new simulations
- Easier to test by creating a mock simulation
- Re-use I/O components in new types of agents

Create a Simulation Abstraction Layer

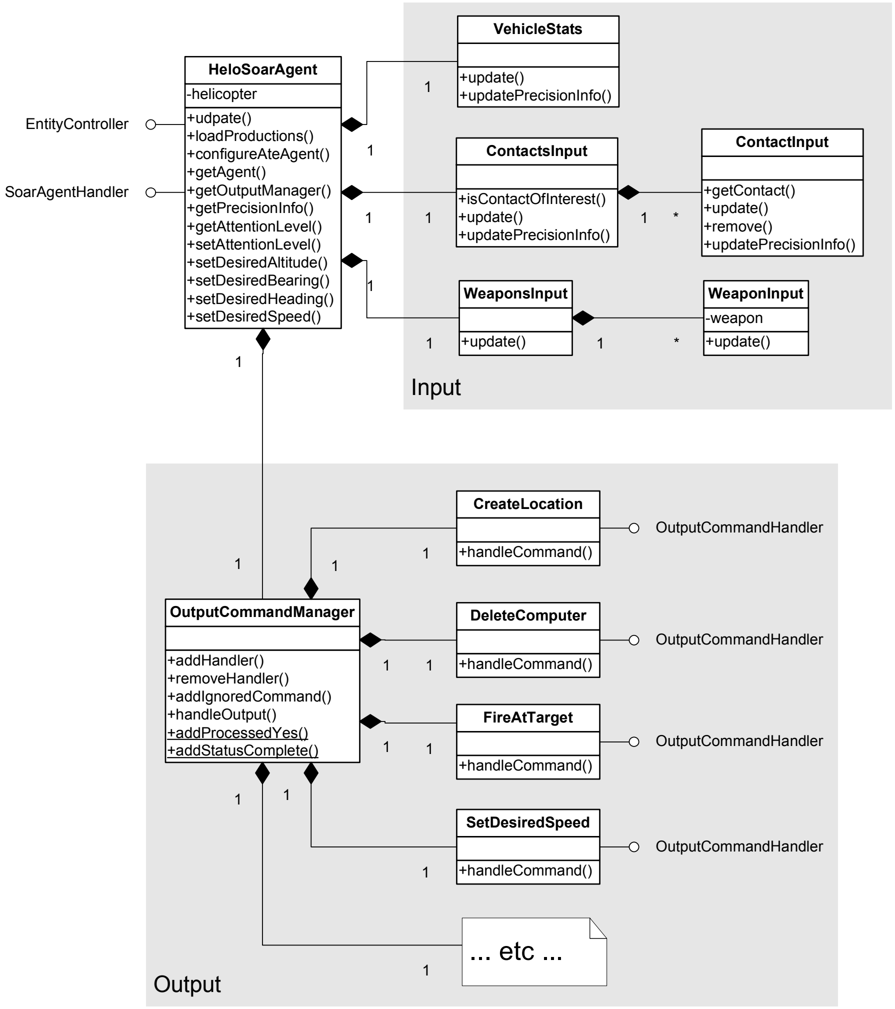


- Log and playback simulation state for testing/debugging

Soar Technology
Thinking *inside* the box.

Think about I/O modularity

- Create separate class for each logical unit of I/O
- One class per input-link structure
- One class per output-link command



EntityController

SoarAgentHandler

HeloSoarAgent
-helicopter
+udpate()
+loadProductions()
+configureAteAgent()
+getAgent()
+getOutputManager()
+getPrecisionInfo()
+getAttentionLevel()
+setAttentionLevel()
+setDesiredAltitude()
+setDesiredBearing()
+setDesiredHeading()
+setDesiredSpeed()

VehicleStats
+update()
+updatePrecisionInfo()

ContactsInput
+isContactOfInterest()
+update()
+updatePrecisionInfo()

ContactInput
+getContact()
+update()
+remove()
+updatePrecisionInfo()

WeaponsInput
+update()

WeaponInput
-weapon
+update()

Input

OutputCommandManager
+addHandler()
+removeHandler()
+addIgnoredCommand()
+handleOutput()
+addProcessedYes()
+addStatusComplete()

CreateLocation
+handleCommand()

OutputCommandHandler

DeleteComputer
+handleCommand()

OutputCommandHandler

FireAtTarget
+handleCommand()

OutputCommandHandler

SetDesiredSpeed
+handleCommand()

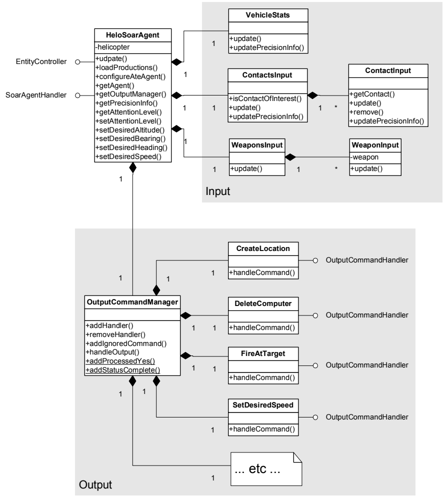
OutputCommandHandler

... etc ...

Output

Think about I/O modularity

- Easier to maintain
  - Where is "set-desired-speed" handled?
  - Oh, in SetDesiredSpeed.java
  - How is "contacts" input structure created?
  - Oh, in ContactsInput.java
- Easier to test
  - I/O classes are decoupled so they can be used in isolation
- Easier to reuse
  - I/O classes are decoupled so they can be dropped in to new agents



Test your Soar I/O code

- Benefits of unit testing
  - Automated – run with continuous integration tools like CruiseControl
  - Easily testable code is more modular
  - Writing tests first forces you to actually use API you’re creating … hopefully a friendlier API results
  - Confidence that new functionality works and didn’t break old code
- Most effective during implementation, **not** after
- How can we unit test our Soar I/O code?
- Here I focus on JUnit but same principles apply to all xUnit-style frameworks

JUnit Basics

- Test case
  - Java class that implements one or more unit tests
  - Unit tests are public methods that start with “test” (JUnit 3)
- setUp()
  - Method called before each unit test is run
  - Initialize objects used by all unit tests in test case
- tearDown()
  - Method called after each unit test is run
  - Called even if test fails
  - Clean up objects initialized in setUp()
- Assertions
  - Assert that the software is in a particular state
  - e.g. assertTrue(passedFunction.wasCalled())

Technicalities

- These are technically ***integration*** tests, not unit tests
- They test both Soar and Java I/O code
- In pure TDD, one or the other would be replaced by a *mock* object
- This is too painful, so we ignore the TDD zealots and call them unit tests anyway

Unit Testing Soar Input

- Use Soar rules to test that input is correct
  - Powerful pattern matching
  - Easier than parsing “print” output (even in XML)
  - Scriptable in Soar if you have Tcl ☺
- Ideally, behavior developer create Soar tests
  - Ensures that behavior developer and software engineer agree on I/O spec

Unit Testing Soar Input

- Create agent with “passed” and “failed” RHS functions
- Initialize Java input class to be tested
- Load productions that test for expected input and call “passed” function
- Run the agent a few steps
- Check that “passed” function was called

Unit Testing Soar Input

- Java side of unit test

ContactsInputTest.java

```
public void setUp()
{
   sim = ...; // Initialize mock simulation with single contact
   agent = ...; // Initialize Soar agent
}

public void testContactAppearsOnInputLink()
{
   // agent and sim initialized in setUp()
   // Install "passed" and "failed" RHS functions
   TestRhsFunctions testFunctions = new TestRhsFunctions(agent.GetKernel());

   // Install the input class we're testing
   ContactsInput contacts = new ContactsInput(sim, agent);

   // Load test productions
   agent.LoadProductions("test/com/soartech/simjr/helosoar/ContactsInputTest.soar");
   SoarException.throwOnError(agent);
   agent.ExecuteCommandLine("run 1");
   assertTrue(testFunctions.passed());
}
```

Unit Testing Soar Input

- Soar side of unit test

ContactsInputTest.soar

```
sp {test*contact
   (state <s> ^superstate nil
              ^io.input-link.contacts <contacts>)
   (<contacts> ^contact <c>  -^contact {<other> <> <c> } )
   (<c> ^callsign test-contact
        ^force red
    ... Test all attributes of <c> ...
-->
   (exec passed)
}

... Test failure conditions ...
```

Unit Testing Soar Output

- Similar to input
- Procedure
  - Create agent
  - Load productions that trigger output command
  - Check that the output command was triggered
  - Check that the output command performed correct actions

Multi-step Unit Tests

- What about multi-step unit tests?
- Test productions may fire in wrong step
- Create a TestStepInput class to put ^test-step on input-link.

```
ContactsInputTest.java

public void testContactAppearsOnInputLink()
{
   ... Other initialization ...
   TestStepInput testStep = new TestStepInput(agent);

   ... First step ...
   testStep.set("initial-contact");
   agent.ExecuteCommandLine("run 1");
   assertTrue(testFunctions.passed());

   testStep.set("contact-destroyed");
   testFunctions.reset();

   contact.setDestroyed(true);
   agent.ExecuteCommandLine("run 1");
   assertTrue(testFunctions.passed());
}
```

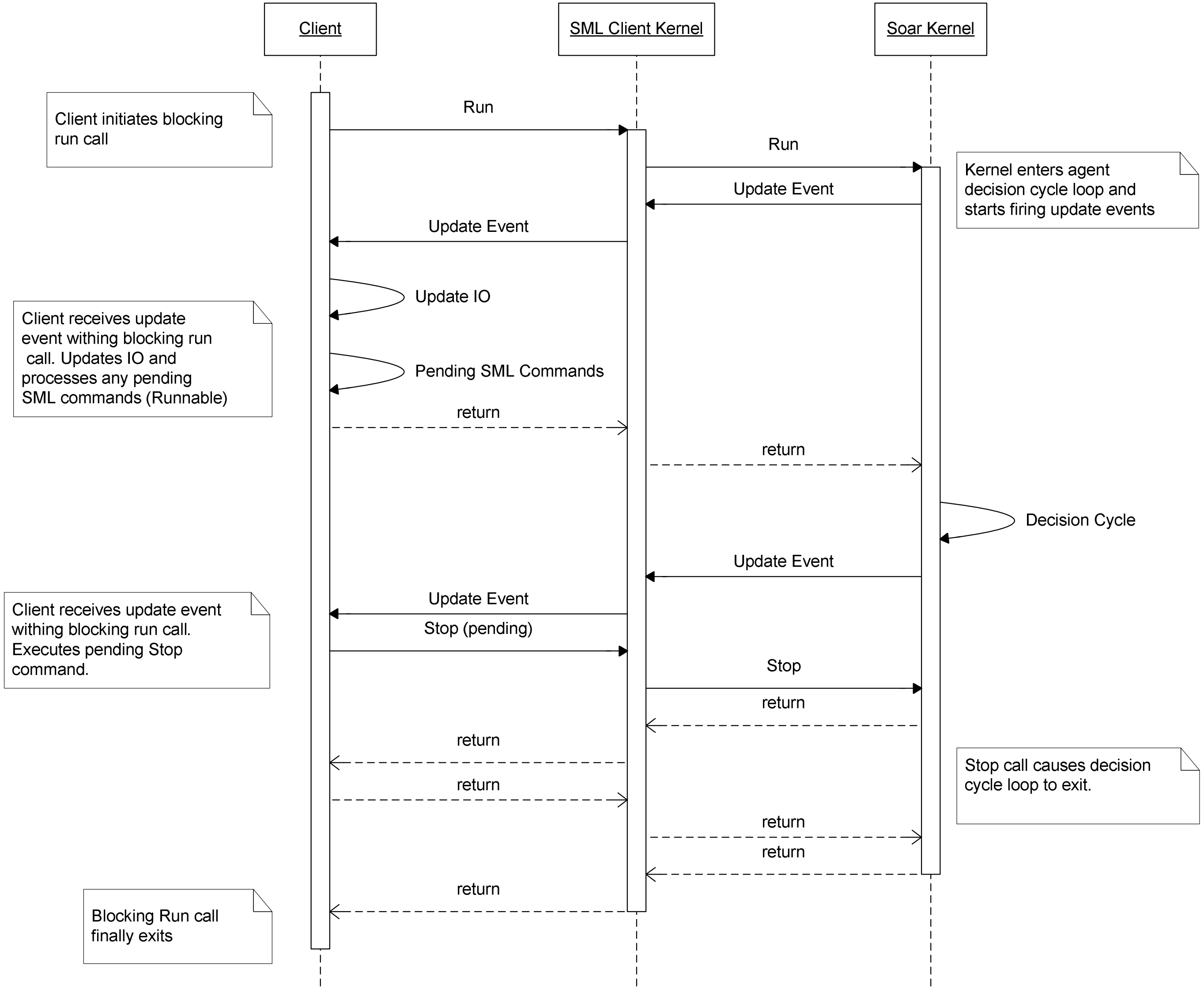
Using SML with JUnit

- Global state accumulates between tests
- “Shutdown()” SML after each test
- My solution
  - Create a custom base class for all test classes
  - Call Shutdown() from tearDown() method

Think about SML threads

- SML Event Thread
  - Receives SML events when client is not running Soar
- SML Run Thread
  - Whatever thread the client calls Run from
  - Run is a blocking call
  - Receives SML events during the Run call
- Basic rules while Soar is running
  - Make SML calls (start and stop, WME creation, etc) **only** from the Run thread
  - In other words, only make calls from callbacks

Think about SML threads (cont)



Client initiates blocking run call

Kernel enters agent decision cycle loop and starts firing update events

Client receives update event withing blocking run call. Updates IO and processes any pending SML commands (Runnable)

Client receives update event withing blocking run call. Executes pending Stop command.

Stop call causes decision cycle loop to exit.

Blocking Run call finally exits

Think about SML threads(cont)

- Remember
  - SML callbacks arrive on event thread or thread Run was called from!
  - All SML commands (run, sp, matches, etc) are **BLOCKING**
  - Only make SML calls from callbacks when running Soar
- If you don’t follow these guidelines
  - Deadlock
  - Corrupted data
  - Despair
- SoarJavaDebugger/src/doc/DocumentThread2.java handles many SML threading issues.

Create a set of SML utilities

- SML C++ API fairly usable
- SWIG-generated API doesn't fit as naturally in other languages (Java, C#, etc)
- Create a set of SML utilities to make it easier to use
  - Wrap commands to turn SML errors (agent.HadError(), etc) into exceptions
  - Function to turn list of output commands into native list
  - Functions to convert WME values to desired type
    - static double getDouble(Identifier parent, String attr, double def)
  - etc.
- Maybe these could be rolled into SWIG-generated code for each target language?

SML Wishlist

- gSKI removal
- Support for multiple kernels in one process
  - Currently can't spread agents across cores without using separate processes
- Allow agents to sleep, like OS threads
  - Reduce CPU usage when agents are just waiting for new input
- RHS functions
  - exec with argument list rather than argument string
  - Ability to register "real" RHS functions with local kernel

The End

- Questions?
- Comments?