CS 350 Project Specifications v0.2

This living document specifies the communication interface that interprets English-like text commands to configure and manipulate the architecture. It guides you in the role of programmer for development in Part I and quasi end user for test and evaluation in Part II (previously referred to as Part III). Everything here derives from decisions made throughout the development process this quarter. Most of it was not your responsibility, but it did happen. Lectures will cover some of the decisions.

The Javadoc will be posted shortly. Initially it documents only what is needed here, but the full documentation should be available soon to help understand what the solution consists of. We already know what we started with in April:

Mechatronics Virtual Testbed

> The customer wants a simulation toolkit with a basic user interface to create, configure, and manipulate a hierarchical network of virtual components comprising actuators for semi-realistic movement from one position to another and sensors for monitoring and reporting their states to support conceptual performance analysis of simplistic real-world engineering systems.

Use our development process to make sense of this document. Ask questions.

Part I: Command Parser

Your team is implementing most of the parser that takes text input from the user and instructs the architecture to perform corresponding actions. Before you do anything with your solution, make sure the provided solution runs correctly in your environment.

The first step is to connect the JAR to your own new project. Save the JAR somewhere related to your project. To connect to it in Eclipse, right-click on the project, then select Build Path→Configure Build Path. In the Libraries tab, select Classpath, click Add External JARs, and select the project JAR. In IntelliJ, select File→Project Structure. In the Modules tab, Dependencies tab, click the + symbol, select JARs or Directories, and select the project JAR. If a new JAR is posted, remove the current one from your project and repeat this process with the new one.

When you run your project (with no other contents), you are executing the `Startup` class in the JAR. You should see this statement appear to standard output:

```
STARTUP
PARSE> @CONFIGURE LOG "a.txt" DOT SEQUENCE "b.txt" NETWORK "c.txt" XML "d.txt"
*** RUNNING SOLUTION PARSER IN ARCHITECTURE ***
PARSE> @exit
@EXIT
```

Now add `Startup.java` to your project (from the task link). This is how to start the project for Part I. You can also put tests here or anything else you want.

Run the project with `Startup` as the `main()`. It should use the parser from the JAR still, and the asterisked statement above should still appear.

The next step is to create your own parser class as defined here. Build the skeleton and make sure it works before adding functionality. The class is called `Parser` in package `cs350s22.component.ui.parser`.

The constructor takes an `A_ParserHelper` and the string containing the command to parse:

```
public Parser(A_ParserHelper parserHelper, String commandText)
```

The parse method parses the command text and performs the actions specified in Architecture Binding for the command:

```
public void parse() throws IOException
```

Class `Startup` shows how to set up and use this code. There is no interactivity in this form.

Run the project again. It should use your parser now, and the asterisked statement should not appear. If it still does, then something is not configured properly.

Each command instantiates a new parser. Therefore, you cannot store anything in your parser that needs to be retained across commands. Class `ParserHelper` already provides most of what your solution needs for this storage. If you need anything else, add the code to `ParserHelper.java` provided and make sure it is in the correct place in your project for the package.

This class provides symbol tables for all components you need to keep track of. A symbol table is simply a hash map that uses the identifier of a component as the key and the component itself as the value. In general, `CREATE` commands put something into a symbol table, and `id` fields within commands get something out of one or more symbol tables.

Description

Commands are presented here in the form of uppercase keywords (although they are not case sensitive) and lowercase fields that require some entry from the user.

- `id` is a standard Java identifier, including underscore (e.g., `myID1`). Put its string contents into an `Identifier` object.
- `value` is a standard signed real or integer value (e.g., `1`, `3.14`, `-5`). Put its numerical contents into a Java double or integer as necessary.
- `string` is a standard string delimited by double quotes. No escape characters are supported. Avoid the backslash character in file paths because it may be misinterpreted by your operating system.

Except for command B3, punctuation is not part of commands. Vertical bar indicates logical *or*; asterisk indicates zero or more instances of the preceding term; plus indicates one or more. Square brackets indicate an optional group. Parentheses show grouping. Subscripts appear for reference in the text.

Whitespace, except in strings, does not matter. All text except identifiers is case insensitive.

A comment prefixed with `//` may follow a command or be on its own line.

All identifiers must be unique. The symbol tables already enforce this constraint.

Commands that specify the recipient or recipients of an action have two options based on two lists of identifiers.

- `ids` is defined as `ID[S] id+`
- `groups` is defined as `GROUP[S] id+`

All components that can receive messages have an identifier. If this identifier is specified in `ids`, the component will be called directly (think email address). These components can also be a member of one or more groups, which are also specified by identifiers. If the group is called by its identifier, all members of the group are subsequently called automatically (think mailing list).

Wherever `[ids]` and `[groups]` appear together, at least one identifier is required in at least one of the two lists.

Keywords with singular and plural forms, like `COMPONENT[S]`, need not align with the list count grammatically.

The architecture does most of the error checking on the results of a command that your parser parses. Your parser needs to throw a `RuntimeException` if it cannot parse the command. The messages and delivery mechanism need not be elaborate or particularly user-friendly.

Use only standard Java (11 or higher), no external tools, libraries, grammar builders, etc. Do not use any resources in the parser package in the architecture. Ask if you are unsure.

All tests must start with the `@CONFIGURE` command. See class `Startup`.

Implement the following commands: A1, C1, C2, C3, C4, D1, D2, D3, E1, E2, E3, E4, E5, E6, F1, G1, G2, H1, I1, I2, I3. Teams of two omit all E and F commands.

There are no new commands to implement for Part I. The new ones in green are for use in Part II.

Orange is updated for Part I.

A. Actuator Commands

The actuator command is responsible for creating an actuator and optionally connecting sensors to it. This component is very similar to the proof of concept in Task 4.

1. `CREATE ACTUATOR (LINEAR | ROTARY) id₁ [groups] [SENSOR[S] id₂+] ACCELERATION LEADIN value₁ LEADOUT value₂ RELAX value₃ VELOCITY LIMIT value₄ VALUE MIN value₅ MAX value₆ INITIAL value₇ JERK LIMIT value₈`

Creates an actuator with identifier id_1 with optional membership in groups and optional embedded sensors id_2. The other arguments are:

$value_1$ The acceleration when the actuator starts moving
$value_2$ The deceleration when the actuator stops moving as it arrives normally at the target value
$value_3$ The deceleration when the actuator stops moving when it is commanded to relax and come to a stop now
$value_4$ The maximum velocity in either direction
$value_5$ The minimum value the actuator can assume
$value_6$ The maximum value the actuator can assume
$value_7$ The initial value the actuator assumes
$value_8$ The acceptable abruptness of changes in velocity as a result of being commanded to a new target value

Example:

```
CREATE ACTUATOR LINEAR myActuator0 ACCELERATION LEADIN 0.1 LEADOUT -0.2 RELAX 0.3 VELOCITY LIMIT 5
  VALUE MIN 1 MAX 10 INITIAL 2 JERK LIMIT 3
CREATE ACTUATOR ROTARY myActuator8 SENSORS mySensor3 ACCELERATION LEADIN 0.1 LEADOUT -0.2 RELAX 0.3
  VELOCITY LIMIT 5 VALUE MIN 1 MAX 10 INITIAL 2 JERK LIMIT 3
```

Architecture Binding:

Get the sensors by calling `get()` with `id` on `SymbolTable<A_Sensor>`, then create an `ActuatorPrototype` object with the arguments and add it to `SymbolTable<A_Actuator>`.

B. Controller Commands

Controller commands are responsible for creating a subnetwork.

1. `CREATE CONTROLLER (FORWARDING | NONFORWARDING) id₁ [groups] [DEPENDENCY SEQUENCER id₂] WITH COMPONENT[S] id₃+`

Creates controller `id₁` and connects components `id₃` and an optional dependency sequencer `id₂`. A forwarding controller automatically forwards incoming messages to its components. A nonforwarding controller does not and therefore needs specific Java code to define its behavior. We are not doing this.

Example:

```
CREATE CONTROLLER FORWARDING myController1 WITH COMPONENTS myActuator1
CREATE CONTROLLER NONFORWARDING myController2 GROUPS myControllerGroup1 DEPENDENCY SEQUENCER
  myDependencySequencer2 WITH COMPONENTS myActuator8
```

Architecture Binding:

Get the components by calling `get()` with `id` and the true argument on `SymbolTable<A_Controller>`, `SymbolTable<A_Actuator>`, and `SymbolTable<A_Sensor>`, then add the components to a `MyControllerSlaveForwarding` or `MyControllerSlaveNonforwarding` object with its `addComponents()` method.

2. `CREATE DEPENDENCY SEQUENCER id₁ SEQUENCE[S] id₂+`

Creates dependency sequencer `id₁` from dependency sequences `id₂`. A sequencer is a collection of logical statements (dependency sequences) that are evaluated in order. For example, if the first sequence of two is waiting on a message with identifier `myMessage1`, it does nothing until `myMessage1` is received. The second sequence then becomes active. When all sequences have been processed, the first becomes active again.

Example:

```
CREATE DEPENDENCY SEQUENCER myDependencySequencer2 SEQUENCES myDependencySequence2
  myDependencySequence3
```

Architecture Binding:

Do not implement this command. It needs B.3, which we are not implementing.

3. `CREATE DEPENDENCY SEQUENCE id₁ EXPRESSION expression`
   where `expression is ( '(' expression (AND | OR) expression ')' ) | id₂`

Creates dependency sequence `id₁` out of the recursive logical expression `expression`. This evaluates to true when a message with identifier `id2` is received by the controller.

The parentheses in quotes are required (but not the quotes). There does not need to be whitespace around them.

Example:

```
CREATE DEPENDENCY SEQUENCE myDependencySequence1 EXPRESSION myMessage1
CREATE DEPENDENCY SEQUENCE myDependencySequence4 EXPRESSION ((myMessage1 AND myMessage2) OR
  (myMessage3 AND (myMessage4 OR myMessage5))
```

Architecture Binding:

Do not implement this command. It requires an understanding of context-free grammars, which is beyond the scope of this project or course.

C. Mapper Commands

Mapper commands are responsible for creating mappers that modify the raw value reported directly by a sensor.

1. `CREATE MAPPER id EQUATION PASSTHROUGH`

Creates mapper `id` that does not remap the raw value. This is equivalent to `CREATE MAPPER id EQUATION SCALE 1`.

Example:

```
CREATE MAPPER myMapper EQUATION PASSTHROUGH
```

Architecture Binding:

Create a new `EquationPassthrough` object, provide it to a new `MapperEquation` object, and add the latter to `SymbolTable<A_Mapper>`.

2. `CREATE MAPPER id EQUATION SCALE value`

Creates mapper `id` that remaps the raw value by the linear coefficient `value`.

Example:

```
CREATE MAPPER myMapper EQUATION SCALE 10
```

Architecture Binding:

Create a new `EquationScaled` object with `value`, provide it to a new `MapperEquation` object, and add the latter to `SymbolTable<A_Mapper>`.

3. `CREATE MAPPER id EQUATION NORMALIZE value₁ value₂`

Creates mapper `id` that remaps the raw value onto a percentage scale as defined by the lower limit $value_1$ and upper limit $value_2$.

Example:

```
CREATE MAPPER myMapper EQUATION NORMALIZE 10 20
```

Architecture Binding:

Create a new `EquationNormalized` object with $value_1$ and $value_2$, provide it to a new `MapperEquation` object, and add the latter to `SymbolTable<A_Mapper>`.

4. `CREATE MAPPER id INTERPOLATION (LINEAR | SPLINE) DEFINITION string`

Creates mapper `id` that remaps the raw value based on the comma-delimited interpolation table defined in `string`. Each row defines a point in the two-dimensional graph. The first value is the raw sensor value; the second is its mapped value. `LINEAR` mode does linear interpolation; `SPLINE` does a smoother nonlinear interpolation.

Examples:

```
CREATE MAPPER myMapper INTERPOLATION LINEAR DEFINITION "mapfile.map"
CREATE MAPPER myMapper INTERPOLATION SPLINE DEFINITION "C:/temp/definition.map"
```

Architecture Binding:

Create a new `MapLoader` object with `filename`, call its `load()` method to get an `InterpolationMap`, provide the latter to a new `InterpolatorLinear` or `InterpolatorSpline` object, provide the latter to a new `MapperInterpolation` object, and add the latter to `SymbolTable<A_Mapper>`.

D. Message Commands

Message commands are responsible for sending messages from the master controller at the top-level network to its components or components in subnetworks.

1. SEND MESSAGE PING

Sends a ping message to the master controller, which propagates it recursively throughout the network regardless of whether controllers are forwarding or nonforwarding.

Example:

```
SEND MESSAGE PING
```

Architecture Binding:

Get the command line interface from `ParserHelper`. Create a `MessagePing` and send it through the command line interface with `issueMessage()`.

2. SEND MESSAGE [ids] [groups] POSITION REQUEST value

Sends a request message to the recipients such that their position is expected to go to `value`, if possible. Inappropriate requests (for example, to a sensor) are ignored.

Example:

```
SEND MESSAGE ID myActuator1 POSITION REQUEST 10
SEND MESSAGE GROUPS myActuators1 myActuators2 POSITION REQUEST 20
SEND MESSAGE ID myActuator1 GROUPS myActuators1 myActuators2 POSITION REQUEST 30
```

Architecture Binding:

```
CommandLineInterface cli = _parserHelper.getCommandLineInterface();
A_Message message = new MessageActuatorRequestPosition(as_appropriate);
cli.issueMessage(message);
```

3. SEND MESSAGE [ids] [groups] POSITION REPORT

Sends a request message to the recipients such that they report their current value, if possible. Inappropriate requests (for example, to a controller) are ignored.

Example:

```
SEND MESSAGE ID myActuator1 POSITION REPORT
SEND MESSAGE GROUPS myActuators1 myActuators2 POSITION REPORT
SEND MESSAGE ID myActuator1 GROUPS myActuators1 myActuators2 POSITION REPORT
```

Architecture Binding:

```
CommandLineInterface cli = _parserHelper.getCommandLineInterface();
A_Message message = new MessageActuatorReportPosition(as_appropriate);
cli.issueMessage(message);
```

E. Meta Commands

Meta commands are responsible for configuring and manipulating the architecture itself, not the network it is executing.

1. `@CLOCK (PAUSE | RESUME)`

Pauses or resumes automated updating by the clock.

Example:

```
@CLOCK PAUSE
@CLOCK RESUME
```

Architecture Binding:

Get the system clock through `Clock.getInstance()` and call its `isActive()` method with the specified state.

2. `@CLOCK ONESTEP [count]`

Updates the clock manually either once or optionally `count` times. This is valid only while the clock is paused.

Example:

```
@CLOCK ONESTEP
@CLOCK ONESTEP 5
```

Architecture Binding:

Get the system clock through `Clock.getInstance()` and call its appropriate `onestep()` method with `count`.

3. `@CLOCK SET RATE value`

Sets the clock rate `value` in milliseconds per update.

Example:

```
@CLOCK SET RATE 20
```

Architecture Binding:

Get the system clock through `Clock.getInstance()` and call its `setRate()` method with `value`.

4. `@EXIT`

Exits the system. This must be the last statement; otherwise, log files may not be complete.

Example:

```
@EXIT
```

Architecture Binding:

Call `exit()` in `ParserHelper`.

5. `@RUN string`

Loads and runs the script in fully qualified filename `string`.

Example:

```
@RUN "myfilename.mvt"
```

Architecture Binding:

Call `run()` in `ParserHelper` with `string`.

6. `@CONFIGURE LOG string1 DOT SEQUENCE string2 NETWORK string3 XML string4`

Defines where the output goes for logging and reporting. This must be the first command issued.

To simplify configuration across various operating systems, output goes to your default temp folder. Print the result of `System.getProperty("java.io.tmpdir")` to find it. The filenames provided here are required but ignored. See Part II for details on the output files.

Example:

```
@CONFIGURE LOG string1 DOT SEQUENCE string2 NETWORK string3 XML string4
```

Architecture Binding:

Call `LoggerMessage.initialize()` with `string1` and `LoggerMessageSequencing.initialize()` with `string2` and `string3`. `string4` is not used.

7. `@CLOCK`

Prints the current time to standard output.

Example:

```
@CLOCK
```

8. `@CLOCK WAIT FOR value`

Waits for `value` seconds before processing the next command.

Example:

```
@CLOCK WAIT FOR 1.5
```

9. `@CLOCK WAIT UNTIL value`

Waits until time is at least `value` seconds before processing the next command.

Example:

```
@CLOCK WAIT UNTIL 2.5
```

F. Network Commands

The network command is responsible for creating the top-level network from actuators, sensors, and/or controllers. The network is the top-level network. It automatically provides in `ParserHelper` its own forwarding controller `MyControllerMaster` called `myControllerMaster`.

1. `BUILD NETWORK WITH COMPONENT[S] id+`

Creates the network with components `id`.

Example:

```
BUILD NETWORK WITH COMPONENT myController
BUILD NETWORK WITH COMPONENTS myController myActuator
```

Architecture Binding:

Get the components by calling `get()` with `id` and the true argument on `SymbolTable<A_Controller>`, `SymbolTable<A_Actuator>`, and `SymbolTable<A_Sensor>`, then add the components to the master controller from `getControllerMaster()` in `ParserHelper` with its `addComponents()` method.

Get the network from `ParserHelper` and call its `writeOutput()` method.

G. Reporter Commands

Reporter commands are responsible for creating reporters. The job of a reporter is to inform recipients of the value of a sensor based on a trigger event.

1. `CREATE REPORTER CHANGE id NOTIFY [ids] [groups] DELTA value`

Creates reporter `id` that informs the recipients when the sensor value has changed by at least `value`.

Example:

```
CREATE REPORTER CHANGE myReporter1 NOTIFY IDS myActuator1 myActuator2 DELTA 3
```

Architecture Binding:

Create a new `ReporterChange` object with `ids`, `groups`, and `value` and add it to `SymbolTable<A_Reporter>`.

2. `CREATE REPORTER FREQUENCY id NOTIFY [ids] [groups] FREQUENCY value`

Creates reporter `id` that informs the recipients every `value` updates.

Example:

```
CREATE REPORTER FREQUENCY myReporter6 NOTIFY IDS myActuator1 myActuator2 GROUPS myGroup3 FREQUENCY 4
```

Architecture Binding:

Create a new `ReporterFrequency` object with `ids`, `groups`, and `value` and add it to `SymbolTable<A_Reporter>`.

H. Sensor Commands

The sensor command is responsible for creating a sensor with optional reporters, optional watchdogs, and an optional mapper.

1. `CREATE SENSOR (SPEED | POSITION) id₁ [groups] [REPORTER[S] id₂+] [WATCHDOG[S] id₃+] [MAPPER id₄]`

Creates a sensor that reports either the speed or position of the actuator it monitors. Position comes from the actuator. Speed is based on change in position with respect to change in time.

An optional mapper maps the raw value of the sensor to its mapped value. If no mapper is provided, the mapped value is the same as the raw value.

Optional reporters send messages at specified times with the mapped value of the sensor.

Optional watchdogs verify that the mapped value of the sensor is in compliance.

Example:

```
CREATE SENSOR POSITION mySensor8 GROUP myGroup1 REPORTERS myReporter1 MAPPER myMapper1
CREATE SENSOR POSITION mySensor16 GROUP myGroup1 REPORTERS myReporter1 WATCHDOGS myWatchdog1
  myWatchdog2 MAPPER myMapper1
```

Architecture Binding:

If reporters, watchdogs, or a mapper are specified, get them from `SymbolTable<A_Reporter>`, `SymbolTable<A_Watchdog>`, or `SymbolTable<A_Mapper>`, respectively. Create a new `MySensor` object with the arguments provided and add it to `SymbolTable<A_Sensor>`.

2. `SET SENSOR id VALUE value`

Sets the value of sensor `id` to `value`. This command is valid only for sensors connected directly to the network, not to sensors embedded in actuators. Networked sensors currently have no measuring capabilities (like temperature or pressure) or components to drive them (like an actuator), so the user is responsible for managing their values manually. Communication does not use messaging because this is a placeholder for something more advanced that we do not have.

Example:

```
SET SENSOR mySensor VALUE 35
```

3. `GET SENSOR id VALUE`

Gets the value of sensor `id` and prints it to standard output. This command is valid for all sensors.

Example:

```
GET SENSOR mySensor VALUE
```

I. Watchdog Commands

Watchdog commands are responsible for monitoring the value of a sensor and reporting whether constraints are violated. All commands support these modes for `mode`:

`MODE (INSTANTANEOUS | (AVERAGE [value₁]) | (STANDARD DEVIATION [value₂]))`

- `INSTANTANEOUS` uses the current value. Use `WatchdogModeInstantaneous`.
- `AVERAGE` uses the average of the last $value_1$ values or all values if $value_1$ is omitted. Use `WatchdogModeAverage`.
- `STANDARD DEVIATION` uses the standard deviation of the last $value_2$ values or all values if $value_2$ is omitted. Use `WatchdogModeStandardDeviation`.

The optional grace value defines how many continuous violations are allowed before reporting. Omitting it reports the first violation.

1. `CREATE WATCHDOG ACCELERATION id mode THRESHOLD LOW value₁ HIGH value₂ [GRACE value₃]`

Creates a watchdog that monitors the acceleration of a value. The watchdog triggers if the acceleration is less than $value_1$ or greater than $value_2$.

Example:

```
CREATE WATCHDOG ACCELERATION myWatchdog1 MODE INSTANTANEOUS THRESHOLD LOW 1 HIGH 3
CREATE WATCHDOG ACCELERATION myWatchdog2 MODE AVERAGE THRESHOLD LOW 1 HIGH 3 GRACE 4
```

Architecture Binding:

Create a `WatchdogAcceleration` object with $value_1$, $value_2$, `mode`, and optionally $value_3$ and add it to `SymbolTable<A_Watchdog>` with `id`.

2. `CREATE WATCHDOG (BAND | NOTCH) id mode THRESHOLD LOW value₁ HIGH value₂ [GRACE value₃]`

Creates a watchdog that monitors a value. For `BAND`, the watchdog triggers if the value is less than $value_1$ or greater than $value_2$. For `NOTCH`, the watchdog triggers if the value is greater than $value_1$ and less than $value_2$.

Example:

```
CREATE WATCHDOG BAND myWatchdog1 MODE INSTANTANEOUS THRESHOLD LOW 1 HIGH 3
CREATE WATCHDOG NOTCH myWatchdog2 MODE AVERAGE 10 THRESHOLD LOW 1 HIGH 3 GRACE 4
```

Architecture Binding:

Create a `WatchdogBand` or `WatchdogNotch` object with $value_1$, $value_2$, `mode`, and optionally $value_3$ and add it to `SymbolTable<A_Watchdog>` with `id`.

3. `CREATE WATCHDOG (LOW | HIGH) id mode THRESHOLD value₁ [GRACE value₂]`

Creates a watchdog that monitors a value. For `LOW`, the watchdog triggers if the value is less than $value_1$. For `HIGH`, the watchdog triggers if the value is greater than $value_1$.

Example:

```
CREATE WATCHDOG LOW myWatchdog1 MODE STANDARD DEVIATION THRESHOLD 3 GRACE 4
CREATE WATCHDOG HIGH myWatchdog2 MODE STANDARD DEVIATION 10 THRESHOLD 3 GRACE 4
```

Architecture Binding:

Create a `WatchdogLow` or `WatchdogHigh` object with $value_1$, `mode`, and optionally $value_2$ and add it to `SymbolTable<A_Watchdog>` with `id`.

Part II: System Test and Evaluation

This final part of the project addresses test and evaluation of the provided solution. It focuses on breadth, not depth. The goal is to demonstrate that each unit of functionality reasonably works for at least one representative scenario. A real test plan for a project of this relatively small size could easily expand to hundreds or even thousands of times the size of this part.

Set up this part the same way you did Part I with a new project, a `Startup` class, and the latest JAR. You are encouraged to use separate files and the `@RUN` command to manage tests, but this is not required.

Deliverable

You need to produce one document with all your tests. Tests are stated in the form of requirements. Unless otherwise specified, you may satisfy each however you want. Each must address the following in exactly this form, including the number, in a separate section:

The test designator and title in bold; e.g., **Test A.1: Basic Actuator Creation**

1. The rationale behind the test; i.e., what is it testing and why we care.
2. A general English description of the initial conditions of the test.
3. The commands for (2), which must appear in a standalone form that could be directly copied into a text file to reproduce the test without manual intervention. Do not cross-reference other tests.
4. A brief English narrative of the expected results of executing the test. (Proper testing discipline expects that you do this *before* running the test.)
5. At least one representation of the actual results. The form is your choice.
6. A brief discussion on how the actual results differ from the expected results.
7. A suggestion for how to extend this test to cover related aspects not required here.

Your document must be formatted professionally. It must be consistent in all respects across all team members. Code references must be in monospaced font.

Tests

Each test is independent. Not all tests may work as expected. Teams of three choose nine tests from this set; teams of two choose six.

Use the various output options to demonstrate that the desired result was achieved. Make it clear in (5) where the answer is.

A. Actuator Tests

Test A.1: Basic Actuator Creation

Create a linear actuator with configuration `LEADIN 0.1 LEADOUT -0.2 RELAX 0.3 VELOCITY LIMIT 5 VALUE MIN 1 MAX 20 INITIAL 2 JERK LIMIT 3`.

Test A.2: Basic Actuator Manipulation

Demonstrate that the actuator from A.1 moves from the initial position to target position 15.

B. Sensor Tests

Task B.1: Standalone Sensor

Create a position sensor and add it to the network. Set and get its value.

Task B.2: Embedded Sensor

Create a position sensor and add it to the actuator from A.1. Set and get its value.

C. Mapper Tests

For each, add the mapper to `mySensor2` from B.2. Show that the raw value from the sensor is appropriately mapped.

Task C.1: Passthrough Mapper

Create a passthrough mapper.

Task C.2: Scaled Mapper

Create a scaled mapper.

Task C.3: Normalized Mapper

Create a normalized mapper.

Task C.4: Interpolation Mapper

Create a spline interpolation mapper with a mapping file of your choice.

D. Reporter Tests

For each, show the reporter reporting and not reporting.

Task D.1: Change Reporter

Create a change reporter.

Task D.2: Frequency Reporter

Create a frequency reporter.

E. Watchdog Tests

For each, show a valid and an invalid case.

Task E.1: Instantaneous Band Watchdog

Demonstrate an instantaneous band watchdog.

Task E.2: Instantaneous Acceleration Watchdog

Demonstrate an instantaneous acceleration watchdog.

Task E.3: Average Low Watchdog

Demonstrate an average low watchdog.

F. Message Tests

Task F.1: Ping Message

Send a ping message to the actuator in A.1.

Output

Most output goes to standard output formatted as follows:

`STARTING` | the system is starting
`EXITING` | the system is exiting
`READ` | a command read from a file
`OUTPUT` | general output
`WATCHDOG` | watchdog output
`TIME` | the current time
`SCHEDULE` | a command scheduled for execution
`EXECUTE` | a command executed

There are four types of output files written to your temp folder, which may be useful in showing test results.

The `project-actuator-`*`id`*`.csv` files log the state of each actuator by *`id`*. Use them to plot Excel graphs.

time	position	velocity	comment
0.33	2	0	StateAscendingLeadin
0.34	2.1	0.1	StateAscendingLeadin
0.35	2.3	0.2	StateAscendingLeadin
0.36	2.6	0.3	StateAscendingLeadin
0.37	3	0.4	StateAscendingLeadin
0.38	3.5	0.5	StateAscendingLeadin
0.39	4.1	0.6	StateAscendingLeadin
0.40	4.8	0.7	StateAscendingLeadin
0.41	5.6	0.8	StateAscendingLeadin
0.42	6.5	0.9	StateAscendingLeadin
0.43	7.5	1	StateAscendingLeadin
0.44	8.6	1.1	StateAscendingLeadin
0.45	9.8	1.2	StateAscendingLeadin
0.46	11.1	1.3	StateAscendingLeadin
0.47	12.5	1.4	StateAscendingLeadin
0.48	14.4	-1.4	StateAscendingLeadin
0.49	15.8	-1.2	StateAscendingLeadout
0.50	17	-1	StateAscendingLeadout
0.51	18	-0.8	StateAscendingLeadout
0.52	18.8	-0.6	StateAscendingLeadout
0.53	19.4	-0.4	StateAscendingLeadout
0.54	19.8	-0.2	StateAscendingLeadout
0.55	20	0	StateAscendingLeadout

The `project-network.xml` file contains an XML representation of the creational and structural aspects of the entire network. It is a snapshot of the initial configuration that is generated only once when `BUILD NETWORK` executes.

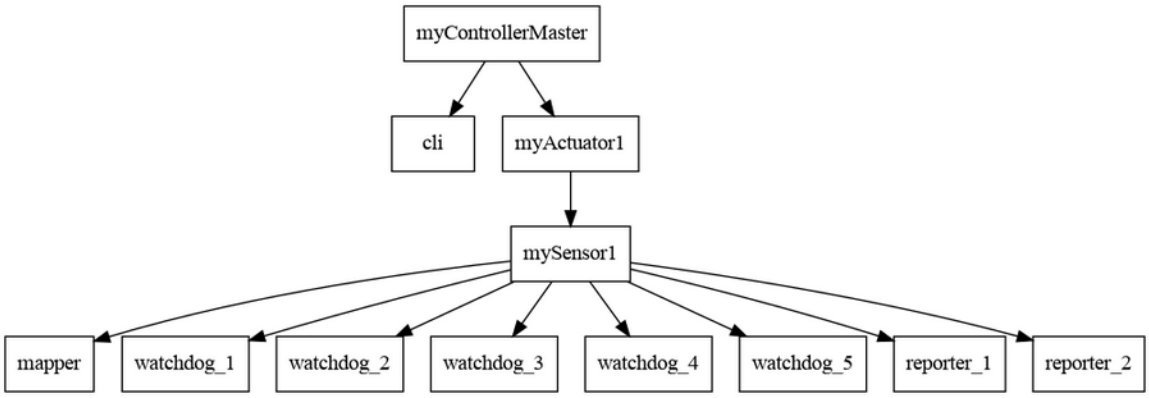
This example (from Chrome) shows an actuator with a heavily loaded sensor:

```
<network>
 <controller id="myControllerMaster" type="MyControllerMaster">
  <components>
   <component id="cli" controllerID="myControllerMaster">
    <groups>
     <group id="interface"/>
     <group id="all"/>
    </groups>
   </component>
   <actuator id="myActuator1" controllerID="myControllerMaster">
    <groups>
     <group id="actuator"/>
     <group id="all"/>
    </groups>
    <sensor id="mySensor1" controllerID="myControllerMaster">
     <groups>
      <group id="sensor"/>
      <group id="all"/>
     </groups>
     <mapper type="interpolator">
      <interpolator type="spline">
       <map>
        <entry valueIndependent="0.0" valueDependent="5.0"/>
        <entry valueIndependent="10.0" valueDependent="100.0"/>
        <entry valueIndependent="20.0" valueDependent="200.0"/>
        <entry valueIndependent="25.0" valueDependent="500.0"/>
        <entry valueIndependent="30.0" valueDependent="400.0"/>
        <entry valueIndependent="40.0" valueDependent="800.0"/>
        <entry valueIndependent="50.0" valueDependent="50.0"/>
       </map>
      </interpolator>
     </mapper>
     <watchdogs>
      <watchdog type="acceleration" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="acceleration" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="acceleration" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="acceleration" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="acceleration" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="band" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="band" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="band" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="band" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="band" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="notch" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="notch" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="notch" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="notch" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="notch" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
      <watchdog type="low" threshold="0.0" grace="3"/>
      <watchdog type="low" threshold="0.0" grace="3"/>
      <watchdog type="low" threshold="0.0" grace="3"/>
      <watchdog type="low" threshold="0.0" grace="3"/>
      <watchdog type="low" threshold="0.0" grace="3"/>
      <watchdog type="high" threshold="0.0" grace="3"/>
      <watchdog type="high" threshold="0.0" grace="3"/>
      <watchdog type="high" threshold="0.0" grace="3"/>
      <watchdog type="high" threshold="0.0" grace="3"/>
      <watchdog type="high" threshold="0.0" grace="3"/>
     </watchdogs>
     <reporters>
      <reporter type="change" deltaThreshold="2"/>
      <reporter type="frequency" reportingFrequency="3"/>
     </reporters>
    </sensor>
   </actuator>
  </components>
 </controller>
</network>
```

The `project-network.dot` file is a Graphviz directed graph of the network in `project-network.xml`. It shows how an MVC architecture can support multiple views from the same model. Graphviz (www.graphviz.org) is a wonderful cross-platform tool you are encouraged to learn. Gnuplot (www.gnuplot.info) is another example. We do not use it here because there is no need for a three-dimensional representation, but examples in lecture showed it in action.

This figure omits watchdogs 6 through 25 because the graph would be huge. Reporters and watchdogs have identifiers in the commands for cross-referencing, but not in the network itself because they are never referred to, so they are shown with arbitrary numerical indices. For logging purposes, they should have identifiers.

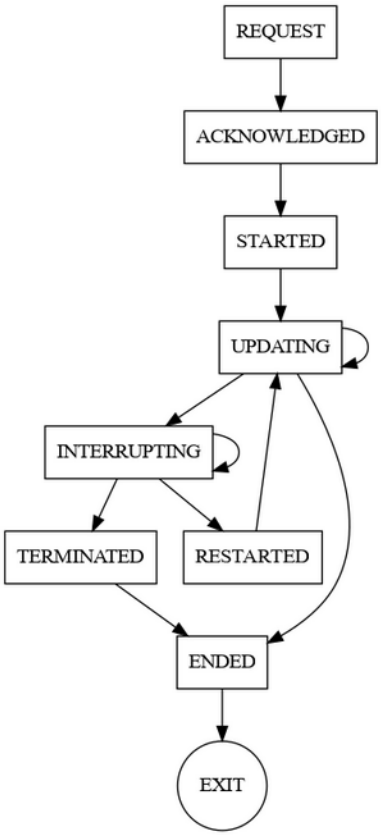
The root component, `myControllerMaster`, is always present in a network, as is `cli` for the command line interface. Even though we are not using it directly, it is still there processing the commands as if the user had actually typed them.



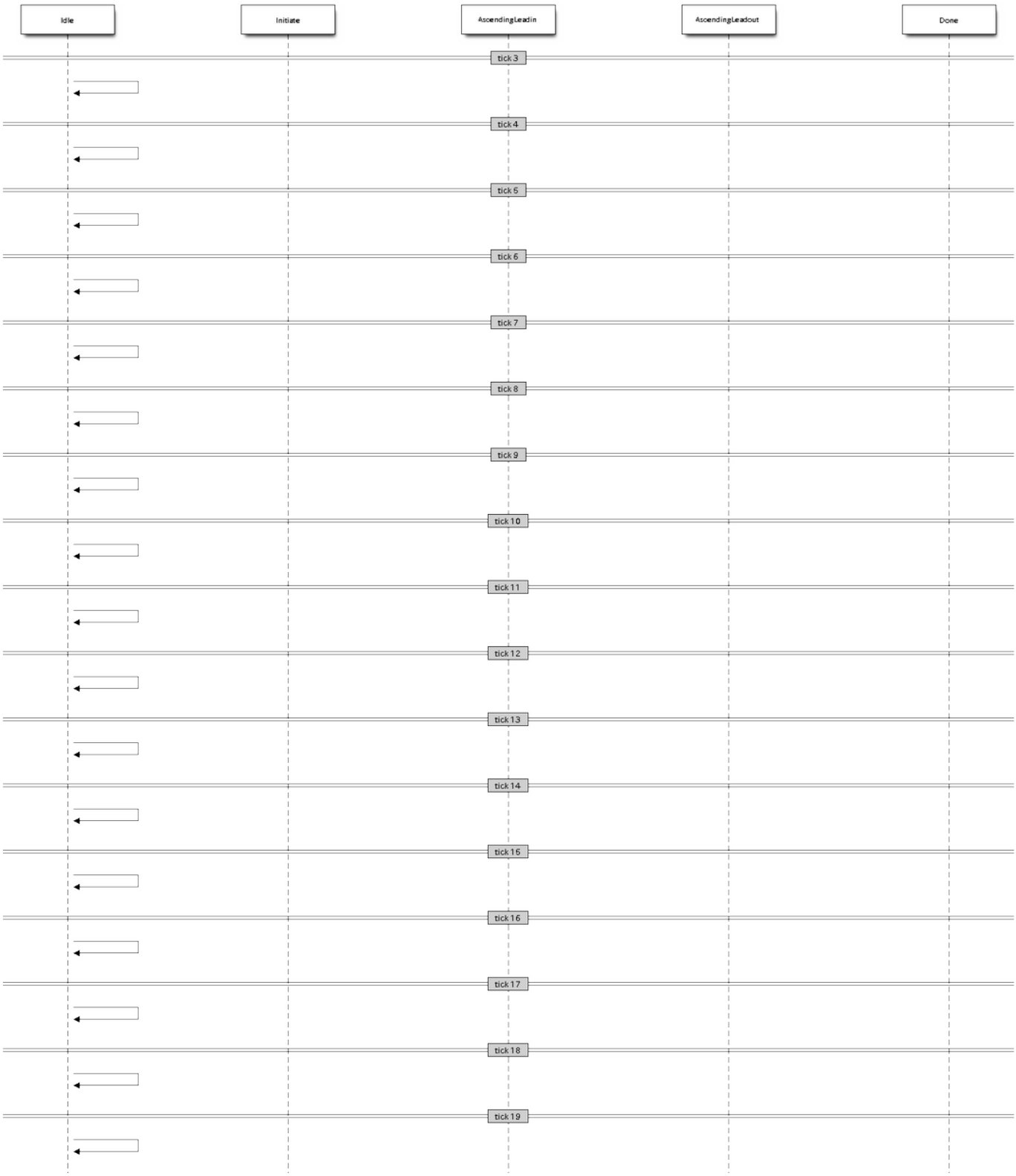
The `project-messages.csv` file is a log of messaging activity on the network:

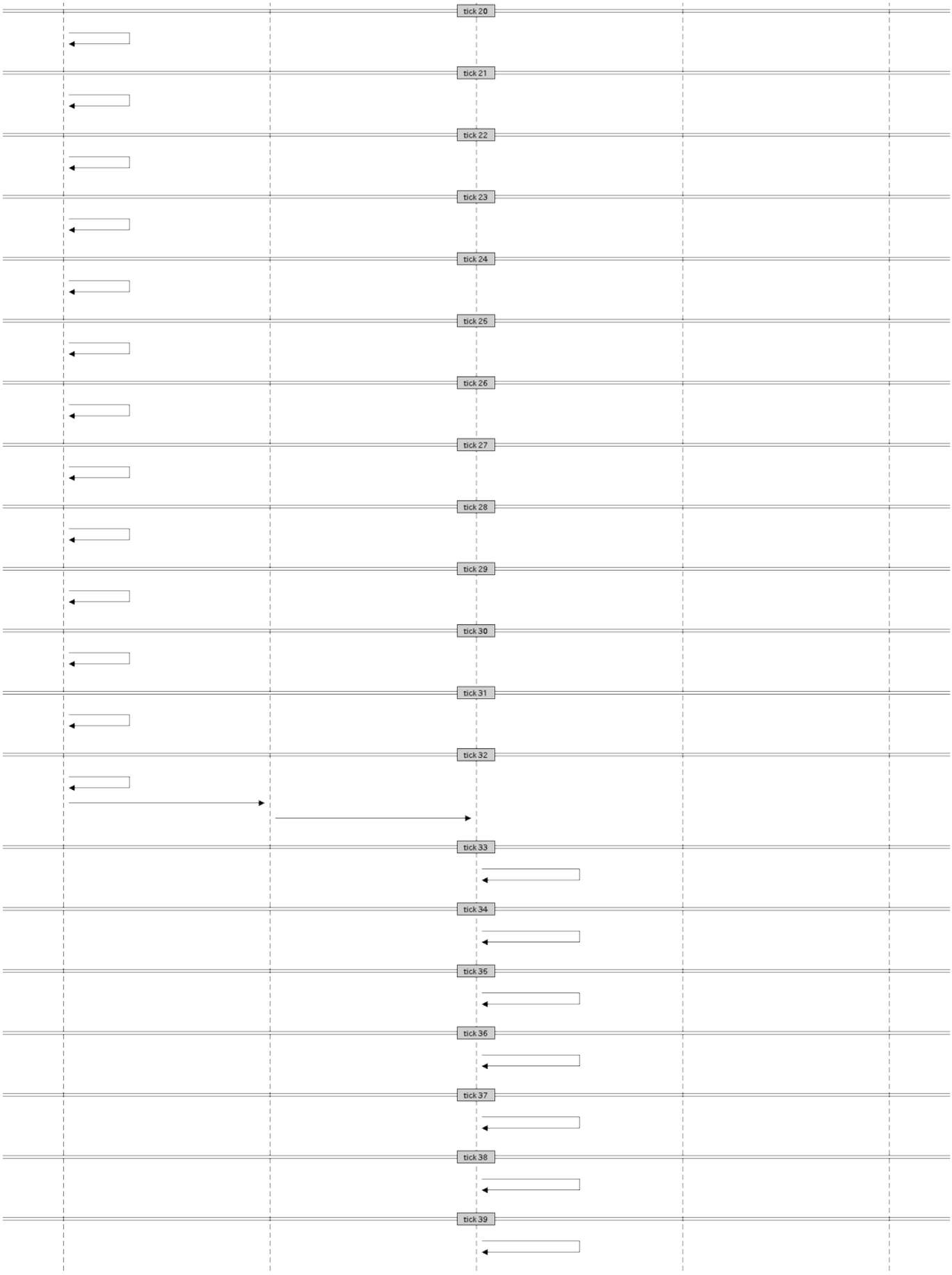
index	time	tick	submitted_tick	message_id	message_type	priority	mode	sender_id	recipient_ids	recipient_groups	recipient_serviced_ids	payload
1	0.02	2	2	1	MessagePing	HIGH	CONTINUE	REQUEST	cli			all
2	0.02	2	2	2	MessagePingReply	NORMAL	CONTINUE	REQUEST	myActuator1		cli	
3	0.22	22	22	167	MessageActuatorReportPosition	NORMAL	CONTINUE	REQUEST	cli		myActuator1	
4	0.22	22	22	177	MessageSensorReportValue	NORMAL	CONTINUE	REQUEST	myActuator1		cli	
5	0.32	32	32	268	MessageActuatorRequestPosition	NORMAL	CONTINUE	REQUEST	cli		myActuator1	
6	0.34	34	34	297	MessageActuatorReportPosition	NORMAL	CONTINUE	REQUEST	cli		myActuator1	
7	0.34	34	34	308	MessageSensorReportValue	NORMAL	CONTINUE	REQUEST	myActuator1		cli	
8	0.45	45	45	435	MessageActuatorReportPosition	NORMAL	CONTINUE	REQUEST	cli		myActuator1	
9	0.45	45	45	446	MessageSensorReportValue	NORMAL	CONTINUE	REQUEST	myActuator1		cli	

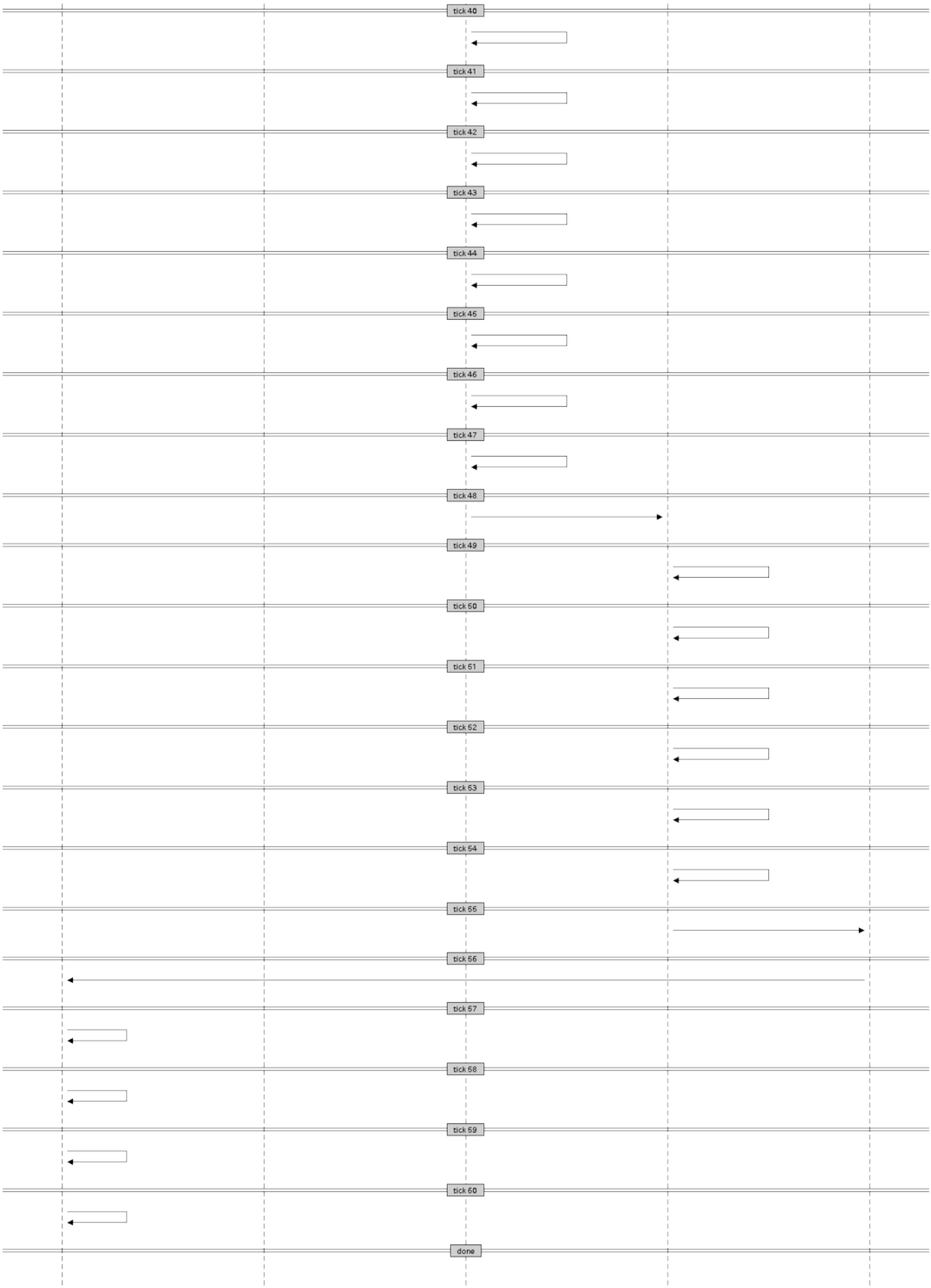
The `project-actuator-id.dot` files are Graphviz sequence diagrams for the state activity of each actuator by *id*. They are based on this state diagram, which defines the road map for the stories a component can execute:



These states occur within the UPDATING state above: idle means nothing is happening; initiate starts the movement process; leadin is acceleration; leadout is deceleration; done completes the process.




tick 20
tick 21
tick 22
tick 23
tick 24
tick 25
tick 26
tick 27
tick 28
tick 29
tick 30
tick 31
tick 32
tick 33
tick 34
tick 35
tick 36
tick 37
tick 38
tick 39


tick 40
tick 41
tick 42
tick 43
tick 44
tick 45
tick 46
tick 47
tick 48
tick 49
tick 50
tick 51
tick 52
tick 53
tick 54
tick 55
tick 56
tick 57
tick 58
tick 59
tick 60
done