Roboliq

Ellis Whitehead

Contents

Preface **5**
Structure of the manual . . . 5
Additional documentation . . . 5
Software information and conventions . . . 5
Acknowledgements . . . 6

1 Introduction **7**
1.1 Motivation . . . 7
1.2 Get started . . . 7
1.3 Usage . . . 7

2 Quick Start **9**
2.1 The simplest protocol . . . 9
2.2 A protocol with a minimal configuration . . . 10
2.3 A BSSE protocol for mario . . . 12

3 Input formats **15**

4 Robot configuration **17**
4.1 Evoware configuration . . . 17
4.2 Components . . . 19
4.3 Logic components . . . 21
4.4 Conclusion . . . 23

5 Simple Protocols **25**
5.1 Objects . . . 26
5.2 Steps . . . 26
5.3 Conclusion . . . 27

6 Advanced Protocols **29**
6.1 Parameters . . . 29
6.2 Objects . . . 29
6.3 Data . . . 30
6.4 Scope . . . 33
6.5 Substitution . . . 33

7 Design Tables **37**
7.1 First examples . . . 37
7.2 Nested branching . . . 39
7.3 Hidden factors . . . 41
7.4 Actions . . . 42
7.5 Step and data nesting . . . 44

Preface

This manual will show you how to write high-level protocols that can be executed on multiple robotic platforms. We introduce *Roboliq*, a software system for automating lab protocols using liquid handling robots. Roboliq support high-level commands that make it much easier to write automation protocols, especially complex ones. It then generates the low-level commands that for your robot system (currently only Tecan Evoware).

Structure of the manual

- Chapter 1 "Introduction" gives a quick overview of how to use Roboliq.
- Chatper 2 "Quick start" is a practical walk-through for writing Roboliq protocols and compiling them.
- Chapter 3 "Input formats" describes several formats that Roboliq accepts for writing protocols.
- Chapter 4 "Robot configuration" describes how to write robot configurations that Roboliq uses to turn high-level commands into low-level instructions.
- Chapter 5 "Simple Protocols" describes the contents of simple Roboliq protocols.
- Chapter 6 "Advanced Protocols" describes the features available for more sophisticated protocols.
- Chapter 7 "Design tables" describes a how to represent experimental designs in a tree or table format, and how these can be used to concisely define complex experiments.

Additional documentation

Roboliq has several other documentation sources that might be of interest to you:

- Protocol – reference documentation for the commands and type available in Roboliq protocols
- Processor API – programmer documentation for Roboliq's protocol processor
- Evoware API – programmer documentation for Roboliq's Evoware backend

Software information and conventions

Roboliq accepts several input formats, but in this manual we will use YAML. YAML is a very popular format for writing software configuration files. If you are not familiar with it yet, please checkout this page on Wikipedia.

Some of the advanced programmable features of Roboliq are demonstrated with JavaScript. Most users won't require knowledge of JavaScript, but if you do need it, there are many good tutorials online.

Acknowledgements

Many many many thanks to Prof. Joerg Stelling for his guidance during my PhD research developing Roboliq. Fabian Rudolf was instrumental in determining which features are most relevant for biologists. Michael Kaltenbach provided sage statistical advice while developing the most sophisticated protocols we tested. Daniel Meyer and Urs Senn provided critical technical support in the lab. Oskari Vinko, Charlotte Ramon and Elena Karamaioti were wonderfully helpful guinnea pigs when testing Roboliq's usability.

Chapter 1

Introduction

1.1 Motivation

Roboliq aims to make it easier to use liquid handling robots for automation in biological laboratories.

It lets you write protocols that are portable between labs, and it compiles the protocols for execution by liquid handling robots.

The only supported backend is for Tecan Evoware robots, but other backends can also be added.

1.2 Get started

This guide presumes some familiarity with the command line terminal. You can install Roboliq by following these steps:

1. Install `nodejs`, which lets you execute Javascript programs.
2. If you're using Microsoft Windows, install the cygwin terminal, which will be used for typing in commands.
3. Copy the Roboliq repository to your computer.
4. Open the terminal, navigate to the directory where you copied the Roboliq repository, and run `npm install` to download Roboliq's software requirements.

Furthermore, the unit tests can be run by running by running `npm test`, and the documentation can be generated by running `npm run docs`.

1.3 Usage

The typical steps in running a Roboliq protocol on a Tecan Evoware robot are:

1. Write a configuration for your lab or use one which has been written by someone else.
2. Write a script for your protocol using the Roboliq syntax and commands.
3. Compile your script and configuration to create a program (.ESC) to be executed by your robot.
4. Open the Tecan Evoware script (.ESC) in Evoware, and run it.
5. Load Roboliq's measurement data into R or another statics program to analyze the data.

Chapter 2

Quick Start

Let's get started with a hands-on walk through. We'll begin with a very basic protocol and build up from there as follows:

1. a protocol that doesn't require any configuration or backend
2. a protocol with a minimal configuration
3. a protocol with a backend
4. a protocol compiled for Evoware
5. running the server and getting the data during execution of an Evoware script

2.1 The simplest protocol

We'll test a simple protocol to show you how to run the software.

1. Navigate to the `protocols` subdirectory and type this code into a file named `walkthrough1.yaml` (or copy the file `walkthrough1-sample.yaml`):

   ```
   roboliq: v1
   description: test that doesn't require configuration
   steps:
     command: system.echo
     text: Hello, World!
   ```

2. Open the terminal, and navigate to the directory where you copied the Roboliq repository.

   ```
   cd ~/Desktop/Ellis/roboliq/
   ```

3. Run Roboliq from the terminal, passing `walkthrough1.yaml` as input:

   ```
   npm run processor -- protocols/walkthrough1.yaml
   ```

   If there were no errors, you can now find a file named `protocols/walkthrough1.out.json`. The output file contains a lot of information: in addition to your script, it also contains the core configuration data that comes with Roboliq by default. If you open the file, you can find a `steps` property which looks like this:

   ```
   "steps": {
     "1": {
       "command": "system._echo",
       "value": "Hello, World!"
     },
   ```

```
  "command": "system.echo",
  "value": "Hello, World!"
}
```

You can see a substep with the command `system._echo`. Commands that begin with an underscore are "final" instructions that will be passed the the backend without further processing. It may be confusing that the original command (`system.echo`) and its properties appear *after* the subcommand, but this is just an artifact of how JavaScript always prints numeric properties first in JSON data.

An important difference between `system.echo` and `system._echo` is that `system.echo` is a higher level command that can handles variables, whereas `system._echo` just takes the value verbatim.

4. Run the command above again, but this time add the `-P compiled` argument to put the output in a separate `compiled` folder.

```
npm run processor -- protocols/walkthrough1.yaml -P compiled/
```

This will automatically create a subfolder named `compiled/walkthrough1` and it will save the file `compiled/walkthrough1/walkthrough1.out.json`. It's important to have a separate folder for every experiment in order to properly organize measured data for each experimental run.

Now you know how to run the software and write a basic protocol. Let's continue to the next step, where we'll see how to create a basic robot configuration file.

2.2 A protocol with a minimal configuration

The robot configuration file lets you specify the capabilities of a robot. Let's start building such a configuration – this task can be quite technical and complicated, so we'll keep it basic here.

We will write the robot configuration in JavaScript; this way, we can later use helper function that will simplify configuring some of the equipment. Please copy the code below to a file named `config/walkthrough2a-config.js`

```
// This variable lets us export our configuration to Roboliq
module.exports = {
  // The targetted version of Roboliq
  "roboliq": "v1",
  // The configuration objects
  "objects": {
    // The top namespace for things in "our lab"
    "ourlab": {
      "type": "Namespace",
      // The namespace for things on our robot "mario"
      "mario": {
        "type": "Namespace",
        // An object to represent Mario's controller software
        // An `Agent` object executes instructions, such as operating equipment.
        "controller": {
          "type": "Agent"
        },
        // The robot's "arm", used for moving labware
        "transporter1": {
          "type": "Transporter",
        },
        // The namespace for mario's "sites" -- i.e., where it can place the
```

```
        // labware.  In this case, we have just two sites, named P1 and P2.
        "site": {
          "type": "Namespace",
          "P1": { "type": "Site" },
          "P2": { "type": "Site" }
        }
      },
      // Namespace for labware models in our lab
      "model": {
        "type": "Namespace",
        // A 96-well plate model with 8 rows and 12 columns
        "plateModel_96well": {
          "type": "PlateModel",
          "label": "96 well plate",
          "rows": 8,
          "columns": 12
        }
      }
    }
  },
  // The logical predicates for our configuration (see explanation in text below)
  "predicates": [
    // -----------------
    // Transporter logic
    // -----------------
    // Declare a site model siteModel_1
    {"isSiteModel": {"model": "ourlab.mario.siteModel_1"}},
    // Our 96-well plate can be placed on top of siteModel_1
    {"stackable": {"below": "ourlab.mario.siteModel_1", "above": "ourlab.model.plateModel_96well"}},
    // Both sites, P1 and P2, are assigned to siteModel_1
    {"siteModel": {"site": "ourlab.mario.site.P1", "siteModel": "ourlab.mario.siteModel_1"}},
    {"siteModel": {"site": "ourlab.mario.site.P2", "siteModel": "ourlab.mario.siteModel_1"}},
    // The list of sites that we can move labware between directly (i.e. without needing to go through
    {"siteCliqueSite": {"siteClique": "ourlab.mario.siteClique1", "site": "ourlab.mario.site.P1"}},
    {"siteCliqueSite": {"siteClique": "ourlab.mario.siteClique1", "site": "ourlab.mario.site.P2"}},
    // Let Roboliq know that mario's controller can use transporter1 to move
    // labware around on siteClique1.
    // The `program` property provides an additional specification for how the
    // transporter should move or grip the labware (not always necessary)
    {"transporter.canAgentEquipmentProgramSites": {
        "agent": "ourlab.mario.controller",
        "equipment": "ourlab.mario.transporter1",
        "program": "Narrow",
        "siteClique": "ourlab.mario.siteClique1"
    }}
  ]
};
```

The logic for labware transportation is somewhat complex – to understand it, you'll need to know the following concepts:

- site: a location where the transporter can place labware
- site model: basically a list of labware that a site can accept; since some sites will accept the same set of labware, they can share the same "site model".

- stackable: this mean that one thing can be placed on top of another thing. In particular, it specified which labware models can go on top of which site models.
- site clique: a set of sites that that permit direct movement of labware among all members of the clique. This concept is a bit tricky. A simple robot configuration might just have a single clique that includes all of its sites. But more complex configuration may need to prohibit some movements. For example, we might have sites on the left of the bench and sites on the right, which are partially blocked by equipment in the middle. If each side of the bench has its own transporter arm and there is a single site in the middle that both arms can reach, then we could define two clicks: the left clique would contain all sites on the left plus the middle site, and the right clique would contain all sites on the right plus the middle site. This would ensure that Roboliq never generates instructions to directly move a plate from the left side to the right side, but rather first navigates through the middle site.

Now write a script to verify that we can move a plate between the two bench sites. Save this file as `protocols/walkthrough2a.yaml` or use the file `protocols/walkthrough2a-sample.yaml`.

```
roboliq: v1
description: Move plate from site P1 to P2
objects:
  plate1:
    type: Plate
    model: ourlab.model.plateModel_96well
    location: ourlab.mario.site.P1
steps:
  command: transporter.movePlate
  object: plate1
  destination: ourlab.mario.site.P2
```

Process the script by running this command from the terminal:

```
npm run processor -- config/walkthrough2a.js protocols/walkthrough2a.yaml -P compiled/
```

Or if you just want to run the sample files which are already present:

```
npm run processor -- config/walkthrough2a-sample.js protocols/walkthrough2a-sample.yaml -P compiled/
```

2.3 A BSSE protocol for mario

Now we'll run a protocol on the real robot (named mario). In our lab, mario has already been configured.

For this example, we'll simply dispense water into each well of a 96-well plate, seal it, and shake it. Create a new file `~/Desktop/Ellis/roboliq/charlotte01.yaml` and write a script like this:

```
roboliq: v1                                  # version of Roboliq being used
# description of this protocol; the pipe symbol "|"" allows for multi-line text
description: |
  Dispense a liquid into each well of a
  96-well plate, seal it, and shake it.
objects:                                     # the set of materials used in this protocol
  plate1:                                    # an object named "plate1"
    type: Plate                              # which is a type of plate
    model: ourlab.model.plateModel_96_round_transparent_nunc  # whose model is defined in the configura
    location: ourlab.mario.site.P3           # which the user should place at the location "P1"
  waterLabware:
    type: Plate
    model: ourlab.model.troughModel_100ml
    location: ourlab.mario.site.R6
```

```
    contents: [Infinity l, water]
  water:
    type: Liquid
    wells: waterLabware(A01 down D01)
steps:
  1:
    command: pipetter.pipette
    sources: water
    destinations: plate1(all)
    volumes: 40 ul
    cleanBetween: none
  2:
    command: sealer.sealPlate
    object: plate1
  3:
    command: shaker.shakePlate
    object: plate1
    program:
      duration: 10 seconds
```

Make sure your in the directory `~/Desktop/Ellis/roboliq/roboliq-processor`, and run the command:

```
npm start -- ../protocols/charlotte01.yaml
```

Chapter 3

Input formats

Roboliq accepts four input formats:

- JSON
- YAML
- JavaScript Node.js module that outputs an object
- JavaScript Node.js module that outputs a function

JSON stands for JavaScript Object Notation. It is a simple format for encoding software data, and it has gradually become the defacto format of choice for transfering data on the web.

YAML is a JSON-like format that is more legible. Due to its superior legibility, this manual usually displays protocols in YAML.

JavaScript is the lingua-franca of the web, and Node.js is the most popular platform for running JavaScript applications outside of browsers. You probably won't need JavaScript for Roboliq, unless you get into more advanced applications. If you do use JavaScript, your module may either 1) output a JavaScript object (i.e. `module.exports = myOjbect;`) or 2) a function that accepts configuration parameters and returns an object (i.e. `module.exports = function(options) { ... return myOjbect; };`).

There are many good online tutorials for the above formats, so please search for one if you encounter confusion.

Chapter 4

Robot configuration

In order for Roboliq to compile your protocol for your robot setup, it needs to know how the robot is configured. This is the most complicated part of Roboliq – it requires advanced technical knowledge or your robot and some programming skill. Once the configuraiton has been specified, future users don't need to be concerned with it.

So if someone else has configured Roboliq for your robot already, then you should probably just proceed to the next chapter. However, if you need to write the configuration, this chapter is for you.

4.1 Evoware configuration

(If you are not using an Evoware robot, you can skip this section.)

Roboliq supplies a simplified method for configuring Evoware robots. You will need to create a JavaScript file in which you define an `EvowareConfigSpec` object, and then have it converted to the Roboliq protocol format. Your config file will have the following structure:

```
const evowareConfigSpec = {
  // Lab name and robot name
  namespace: "YOUR LAB ID",
  name: "YOUR ROBOT ID",
  // Compiler settings
  config: {
    TEMPDIR: "TEMPORARY DIRECTORY FOR MEASUREMENT FILES",
    ROBOLIQ: "COMMAND TO CALL ROBOLIQ'S RUNTIME",
    BROWSER: "PATH TO WEB BROWSER"
  },
  // Bench sites on the robot
  sites: {
    MYSITE1: {evowareCarrier: "CARRIER ID", evowareGrid: MYGRID, evowareSite: MYSITE},
    ...
  },
  // Labware models
  models: {
    MYPLATEMODEL1: {type: "PlateModel", rows: 8, columns: 12, evowareName: "EVOWARE LABWARE NAME"},
    ...
  },
  // List of which sites and labware models can be used together
  siteModelCompatibilities: [
```

```
  {
    sites: ["MYSITE1", ...],
    models: ["MYPLATEMODEL1", ...]
  },
  ...
],
// List of the robot's equipment
equipment: {
  MYEQUIPMENT1: {
    module: "EQUIPMENT1.js",
    params: {
      ...
    }
  }
},
// List of which lid types can be stacked on which labware models
lidStacking: [
  {
    lids: ["lidModel_standard"],
    models: ["MYPLATEMODEL1"]
  }
],
// List of the robot's robotic arms
romas: [
  {
    description: "roma1",
    // List of sites this ROMA can safely access with which vectors
    safeVectorCliques: [
              { vector: "Narrow", clique: ["MYSITE1", ...] },
              ...
          ]
  },
  ...
],
// Liquid handing arm
liha: {
  // Available tip models
  tipModels: {
    MYTIPMODEL1000: {programCode: "1000", min: "3ul", max: "950ul", canHandleSeal: false, canHandleCe
    ...
  },
  // List of LIHA syringes (e.g. 8 entries if it has 8 syringes)
  syringes: [
    { tipModelPermanent: "MYTIPMODEL1000" },
    ...
  ],
  // Sites that the LIHA can access
  sites: ["MYSITE1", ...],
  // Specifications for how to wash the tips
  washPrograms: {
    // For Example: Specification for flushing the tips with `programCode == 1000`
    flush_1000: { ... },
    ...
```

```
    }
  },
  // Additional user-defined command handlers
  commandHandlers: {
    "MYCOMMAND1": function(params, parsed, data) { ... },
    ...
  },
  // Optional functions to choose among planning alternatives
  planAlternativeChoosers: {
    // For Example: when the `shaker.shakePlate` command has
    // multiple shakers available, you might want to use
    // the one name `MYEQUIPMENT1`
    "shaker.canAgentEquipmentSite": (alternatives) => {
            const l = alternatives.filter(x => x.equipment.endsWith("MYEQUIPMENT1"));
            if (l.length > 0)
                return l[0];
        }
  }
};

const EvowareConfigSpec = require('roboliq-evoware/dist/EvowareConfigSpec.js');
module.exports = EvowareConfigSpec.makeProtocol(evowareConfigSpec);
```

The details about `evowareConfigSpec` are in the Evoware API documenation, and an extensive example can be found in `roboliq-processor/tests/ourlab.js`. This information is probably enough to configure your Evoware robot, and you can skip the rest of this chapter unless you need to setup a different kind of robot.

4.2 Components

A *configuration* is a JavaScript object with the following properties:

- `roboliq`: specifies the version of Roboliq.
- `schemas`: JSON Schema defintitions for all object and commands.
- `objects`: specifies the objects provided by the robot.
- `commandHandlers`: specifies the functions that handle protocol commands, for example by generating low-level commands from high-level commands.
- `predicates`: specifies the *logical predicates* used by Roboliq's A.I. to figure out how to compile some high-level commands to low-level commands for this configuration.
- `objectToPredicateConverters`: specifies functions that generate logical predicates based from `objects`.
- `planAlternativeChoosers`: specifies functions that can choose a specific plan among various alternative plans
- `planHandlers`: specifies the functions that transform logical tasks into commands.

The first four properties are easy for the average JavaScript programmer to understand: they are straight-forward JSON values or JavaScript functions. In contrast, the last four properties (`predicates`, `objectToPredicateConverters`, `planAlternativeChoosers`, and `planHandlers`) rely on concepts from Artificial Intelligence, which will require more effort to grasp.

4.2.1 `roboliq`

As with protocols, the first property of a configuration should be `roboliq: v1`.

4.2.2 schemas

For every command and object type, Roboliq requires a schema that defines its properties. By convention, object types begin with an upper-case letter, and commands begin with a lower-case letter. The schemas are written in a standardized format, which you can learn more about at JSON Schema.

Normally, the schemas required for your robot should be automatically provided by your chosen backend and the equipment you select. The `schemas` property is a hash map: its keys are the command names and object type names; its values are the respective JSON schemas. You will not need to write schemas unless you are creating new commands or objects types for Roboliq. An example schema is provided below in the `commandHandlers` section.

4.2.3 objects

The `objects` property of a configuration is the same as describe for protocols in the Quick Start chapter. However, robot configurations usually contain complex objects, such as measurement devices, whereas protocols usually only define fairly simple labware. Furthermore, the configuration may contain information that's required for the backend compiler. The object schemas are described in the documentation for standard Roboliq objects and for Evoware objects.

Namespaces. Because a robot "contains" its devices, sites, and permanent labware, we use `Namespace` types to build nested objects. For example, if our lab is named "bsse", we have a robot named "bert", and it has a site named "P1", this could be encoded as follows:

```
objects:
  bsse:
    type: Namespace
    bert:
      type: Namespace
      site:
        type: Namespace
        P1:
          type: Site
          ...
```

We can then reference the site in the protocol as `bsse.bert.site.P1`.

4.2.4 commandHandlers

A command handler is a JavaScript function that processes command parameters and returns information to Roboliq about the command's effects, sub-commands, and user messages.

Command handlers are supplied by various modules. Roboliq's command modules are in the subdirectory `roboliq-processor/src/commands`, and Evoware's command modules are in the subdirectory `roboliq-evoware/src/commands`. The API documentation contains information about the available commands.

If you want to write your own command handler, you can add it to `commandHandlers` as a property. The property name should be the name of the command, and the value should be the command handler function. For example, let's make a simplistic function called `my.hello` that tells an Evoware robot to say "Hello, YOURNAME!". First we'll defined the schema for the new command:

```
schemas: {
  "my.hello": {
    description: "Say hello to someone",
    properties: {
```

```
      name: {
        description: "Name to say hello to",
        type: "string"
      }
    },
    required: ["name"]
  }
}
```

This schema lets Roboliq know that there's a command named `my.hello`.

```
commandHandlers: {
  "my.hello": function(params, parsed, data) {
    return {
      expansion: [
                {
                    command: "evoware._userPrompt",
                    text: "Hello, "+parsed.value.name",
                }
            ]
    };
  }
}
```

4.3 Logic components

Roboliq uses an Articifial Intelligence method called Hierarchical Task Network (HTN) Planning. In particular, it uses a SHOP2 implementation written by Warren Sack in JavaScript.

4.3.1 `predicates`

A predicate defines a "true statement". For example, the following predicate is named `sealer.canAgentEquipmentProgramMode` and it lets Roboliq know that the robot agent `ourlab.mario.evoware` can use the sealer `ourlab.mario.sealer` with the labware model `ourlab.model.plateModel_96_round_transparent_nunc` at site `ourlab.mario.site.ROBOSEAL` with the internal program file `PerkinElmer_weiss.bcf`.

```
yaml "sealer.canAgentEquipmentProgramModelSite":     "agent": "ourlab.mario.evoware"
"equipment": "ourlab.mario.sealer"     "program": "C:\\HJBioanalytikGmbH\\...\\PerkinElmer_weiss.bcf",
"model": "ourlab.model.plateModel_96_round_transparent_nunc",     "site": "ourlab.mario.site.ROBOSEAL"ya
```

The general form for predicates is:

```
predicateName:
  object1: value1
  object2: value2
  ...
```

The values are usually object names, and predicates often define true relationships among objects. All of the predicates together form a database of true statements about the "world" in which the protocol will run. They are used by certain command handlers to automatically figure out valid operations without the user needing to specify the low-level details. (Currently, the end-user documentation does not contain details about which predicates are used by which commands, but it can be found in the command handler source code.)

Here is an example excerpt of using the predicate database to find all valid sealers from the `sealer.sealPlate` command handler:

```
        const predicates = [
            {"sealer.canAgentEquipmentProgramModelSite": {
                "agent": parsed.objectName.agent,
                "equipment": parsed.objectName.equipment,
                "program": parsed.objectName.program,
                "model": model,
                "site": parsed.objectName.site
            }}
        ];
        const [chosen, alternatives] = commandHelper.queryLogic(data, predicates, "sealer.canAgentEquip
```

The function `commandHelper.queryLogic()` will find all solutions the `predicates` array, filling in the missing values as necessary. If more than one alternative solution is present, it will choose one of them. The solution can either by chosen by a appropriate function `planAlternativeChoosers`, or else Roboliq simply picks the first item in the alternatives list.

4.3.2 `objectToPredicateConverters`

Roboliq's AI needs predicates describing the available objects. These are generated dynamically by the functions supplied in `objectToPredicateConverters`. It is a map from an object type to a function that accepts a named object and returns an array of predicates. For example, here is Roboliq's converter for `Plate` objects:

```
objectToPredicateConverters: {
  Plate: function(name, object) {
        const predicates = [
            { "isLabware": { "labware": name } },
            { "isPlate": { "labware": name } },
            { "model": { "labware": name, "model": object.model } },
            { "location": { "labware": name, "site": object.location } }
        ];
        if (object.sealed) {
            predicates.push({ "plateIsSealed": { "labware": name } });
        }
        return predicates;
    },
    ...
}
```

This converter creates between 4 and 6 predicates for every plate object: `isLabware` lets the AI know that the plate is a kind of labware, `isPlate` says that the plate is a plate, `model` says which labware model the plate has, and `location` says where the plate is located. Furthermore, `plateIsSealed` will be present if-and-only-if the plate is sealed.

You'll only need to create additional object-to-predicate converters if you want to extend Roboliq's object types, or perhaps if you create an advanced command that required additional logic.

4.3.3 `planHandlers`

Plan handlers are functions that convert from a planning action to a Roboliq command (usually a low-level command). It is a map from an action name to a function that accepts that action parameters and returns

an array of commands. The function also accepts the parameters of the parent command (the one that generated the plan), and the protocol data, in case those are needed to compute the new command.

An example use-case is using the `transporter.movePlate` command to move a plate into a closed centrifuge: the planning algorithm will include an action to open the centrifuge first, and then the `transporter.movePlate` command will call the appropriate function in `planHandlers` to create the required sub-command.

4.4 Conclusion

Configuring a robot can be complicated. First of all, it requires a lot of detailed knowledge about your robot. Secondly, it involves a lot of interdependencies. For example, in order to support a new labware model, you'll need to add the model to the `models` list (easy), but you also need to update the list of site/model compatibilities, the list of safe transport vectors, and perhaps the list of which models accept lids or can be stacked on top of each other.

It's certainly do-able, but you're likely to encounter some frustrations if you need to trouble-shoot why you get compiler errors when trying to use your new model, for example.

Chapter 5

Simple Protocols

Roboliq was developed to help automate protocols on liquid handling robots, especially for lab experiments in molecular biology. This chapter will explain the four properties in simple protocols and how to write write them.

First consider this short example – it uses Roboliq version "v1"; it defines a `Plate` named `sourcePlate`; and it instructs us to shake the plate:

```
roboliq: v1
description: Shake a plate, just because we can
objects:
  sourcePlate:
    type: Plate
steps:
  1:
    command: shaker.shakePlate
    object: sourcePlate
```

The protocol consists of *properties* and *values*. A *property* is a name followed by a semicolon and then a *value*. In the above example, we see the following properties and values:

1) The first property is `roboliq`, and it has a string value of `v1` which indicates we're using Roboliq version 1.

2) The second property is `description`, whose value is also a text string.

3) The third property is `objects`, which defines the materials we'll use in the protocol. In contrast to the previous properties, the *value* starts on the next line and is indented by two spaces. This means that the value of `objects` is another set of properties: `sourcePlate` is the name of a material whose `type` property is set to `Plate`.

4) The forth property is `steps`, which defines the steps to be performed. Its *value* is usually a numbered set of steps, and each numbered step is assigned properties as well: in this case, step `1` has the properties `command: shaker.shakePlate` and `object: sourcePlate` which tell Roboliq to shake the plate named `sourcePlate`.

Most of your protocols will have those four basic properties, and perhaps additional properties as well.

Next we will describe the `objects` and `steps` properties in more detail.

5.1 Objects

In order to add more objects, just give them a name and assign them the appropriate properties. Here we extend `objects` to include another plate for mixing named `mixPlate`:

```
objects:
  sourcePlate:
    type: Plate
  mixPlate:
    type: Plate
```

Objects are the things that can be used in the protocol's steps, including labware, liquids, and equipment. Each object requires a `type` property (e.g. `Plate` in the example above) – the two most common types used in protocols are:

- `Plate`: for defining labware, including tubes and troughs.
- `Liquid`: for defining liquids.

A complete list of types can be found in the Commands & Types documentation, but most of them are only used in robot configurations rather than in protocols.

In addition to the `type` property, object have other properties as well (also available in the type documentation), and all objects have an optional `description` property that you can use to add your own notes about the object. Here are examples of the two main object types to indicate that a liquid named `specialMix` is in all wells on `plate1`:

```
objects:
  plate1:
    type: Plate
    description: Plate to be used for initial mixing
    model: ourlab.roboto.model.96microwell
    location: ourlab.roboto.P1
  specialMix:
    type: Liquid
    description: Our special mad-scientist mix
    wells: plate1(all)
```

5.2 Steps

A *step* is either a command or a numbered list of sub-steps, and a protocol's `steps` property specifies the steps that should be taken during the protocol. Here is an example with two numbered steps in which a plate is sealed and shaken:

```
steps:
  1:
    command: sealer.sealPlate
    object: plate1
  2:
    command: shaker.shakePlate
    object: plate1
```

A *command* is indicated by a `command` property, such as the commands `sealer.sealPlate` and `shaker.shakePlate` above. Besides the `command` property, you will need to specify additional properties to tell the command exactly what to do. In the above example, the line `object: plate1` tells the command to act on `plate1`.

Command names have two parts separated by a period. First is the command category, which contains related commands. Second is the actual command name. Roboliq's standard commands are listed in the Command documentation, and Evoware's special commands are listed in the Evoware Command documentation.

Steps can also be given a `description` property that let's you explanation what's happening for someone else who reads the protocol (or for yourself later after you've forgotten). Step can also be given a `comment` property, which is intended to be used as programmer comments that aren't of interest to others.

Here is an example that contains nested steps, descriptions and commands - it is the same as the previous example, but applied to two plates:

```
steps:
  1:
    description: "Handle plate1"
    1:
      command: sealer.sealPlate
      object: plate1
    2:
      command: shaker.shakePlate
      object: plate1
  2:
    description: "Handle plate2"
    1:
      command: sealer.sealPlate
      object: plate2
    2:
      command: shaker.shakePlate
      object: plate2
```

5.3 Conclusion

Simple protocols can be writen using four properties: `roboliq`, `description`, `objects`, and `steps`. The `roboliq` property indicates the version of Roboliq, and `description` lets you document what the protocol does. In `objects` you specify the labware and liquids used in the protocol. And in `steps` you create a list of numbered commands; the steps can be nested and documented however you prefer. You can find documentation for the objects and commands in the Commands & Types documentation.

Chapter 6

Advanced Protocols

Advanced protocols can contain many more elements than the simple protocols described in the previous chapter. In this chapter we'll discuss `parameters`, `objects`, `data` properties, variable scope, substitution, directives, and template functions.

6.1 Parameters

Parameters are named values that you define at the beginning of a protocol. You can then use the parameter name instead of the value later in the protocol. In this partial example, a `VOLUME` parameter is defined which is used in the `pipetter.pipette` step:

```
parameters:
  VOLUME:
    description: "amount of water to dispense"
    value: 200 ul
...
steps:
  1:
    command: pipetter.pipette
    sources: water
    destinations: plate1(all)
    volumes: $#VOLUME
```

A parameter should be given a `description` and a `value`. Notice that `$#` prefixes `VOLUME` in the step; `$#` tells Roboliq to substitute in a parameter value. Substitution is discussed in more detail later in this chapter.

6.2 Objects

There are several more object types you might want to use in advanced protocols:

`Data`: for defining a data table. The `Data` type facilitates complex experimental designs, and you can read more about it in the next section and in the chapter on Design Tables.

`Variable`: For defining references to other variables. Variables are not particularly useful in Roboliq, but they could potentially be used to easily switch between objects in case you have, for example, two water sources `water1` and `water2`. In that case you could have a `water` variable whose value to set to the source you want to use:

```
objects:
  water1: ...
  water2: ...
  water:
    description: "water source"
    type: Variable
    value: water1
steps:
  1:
    command: pipetter.pipette
    sources: water
    destinations: plate1(all)
    volumes: 50 ul
```

`Template`: You can use a template to define re-usable steps. Here is a toy example for a template named `dispenseToPlate1` which creates a `pipetter.pipette` command that transfers a volume of water to all wells on `plate1`:

```
objects:
  plate1:
    type: Plate
    ...
  dispenseToPlate1:
    description: "template to dispense `volume` ul of water to all wells on plate1"
    type: Template
    template:
      command: pipetter.pipette
      sources: water
      destinations: plate1(all)
      volumes: "{{volume}}"
steps:
  1:
    command: system.call
    name: dispenseToPlate1
    params:
      volume: 10 ul
  2:
    command: system.call
    name: dispenseToPlate1
    params:
      volume: 50 ul
```

This protocol will first dispense 10ul to all wells, then another 50ul to all wells – not actually a useful protocol, but it illustrates the point. In the `system.call` command, the name of the template is specified along with the parameters template parameters. Templates are expanded using the Handlebars template engine, which is the reason for the "{{" and "}}" delimiters in the line `volumes: "{{volume}}"`.

6.3 Data

Roboliq supports data tables to enable complex experiments. Conceptually, a data table is like a spread sheet of rows and named columns, where each row represents some "thing", a each column represents a property. In Roboliq, a data table is an array of JSON objects: each object is a row, and each property is a column. Normally all the objects will have the same set of properties (but this is not required).

Data tables are supported by a `Data` type, a `data` property, a `data()` directive, and a set of `data.*` commands.

6.3.1 `Data` type

You can define a data table using the `Data` object type. Here's an example where each row has a well, a liquid source, and a volume:

```
objects:
  data1:
    type: Data
    value:
      - {well: A01, volume: 10 ul, source: liquid1}
      - {well: B01, volume: 10 ul, source: liquid2}
      - {well: A02, volume: 20 ul, source: liquid1}
      - {well: B02, volume: 20 ul, source: liquid2}
```

You can define as many data tables as you want in a protocol.

6.3.2 `data` property

After defining a data table, you need to "activate" it for usage. This is done using the `data` property, which understands the following parameters:

- `source`: this is the name of a `Data` object.
- `where`: this is an optional boolean mathjs expression that is evaluated for each data row – only rows for which the expression evaluates to true are activated.
- `orderBy`: an optional array of column names for ordering rows. The ordering behavior is the same as the `_.sortBy` function in lodash.

Any step can be given a `data` property to make a table available in that step and its sub-steps. Here's an example application:

```
steps:
  1:
    data: {source: data1}
    command: pipetter.pipette
    sources: $$source
    destinationPlate: plate1
    destinations: $$well
    volumes: $$volume
```

The `data` property activates our data table `data1`. The command `pipetter.pipette` can now access the data columns by using the `$$`-prefix along with the column name. So the above example is essentially equivalent to this:

```
steps:
  1:
    command: pipetter.pipette
    sources: [liquid1, liquid2, liquid1, liquid2]
    destinationPlate: plate1
    destinations: [A01, B01, A02, B02]
    volumes: [10 ul, 10 ul, 20 ul, 20 ul]
```

6.3.3 `data.*` commands

Roboliq's `data.*` commands provide two commands for more sophisticated handling of data tables: `data.forEachRow` and `data.forEachGroup`.

`data.forEachRow` lets you run a series of steps on each row of the data table. For each row, the command activates a new data table containing only that row, and it runs its sub-`steps` using that new table. Here's a toy example:

```
steps:
  1:
    data: {source: data1}
    command: data.forEachRow
    steps:
      1:
        command: pipetter.pipette
        sources: $source
        destinationPlate: plate1
        destinations: $well
        volumes: $volume
      2:
        command: fluorescenceReader.measurePlate
        object: plate1
        output:
          joinKey: well
```

In this case, the sub-steps will be repeated 4 times, once for each row. That mean each well will be dispensed into and measured before moving onto the next well. Notice that here only a single $-prefix was used rather than the double $$-prefix for the column variables. When a data table is activated, Roboliq will check if any of the columns have all the same value; if so, that property and value will be automatically added to the current scope (see the next section about Scope). Scope variables are accessible via the $-prefix. Since the `data.forEachRow` command activates each row individually, all of its columns will be added to the scope.

`data.forEachGroup` lets you operate on groups of rows at a time. You provide a `groupBy` property for it to group by, and then for each group it activates a new data table with those rows and runs its sub-`steps` using that new table. Here's another toy example:

```
steps:
  1:
    data: {source: data1}
    command: data.forEachGroup
    groupBy: source
    steps:
      1:
        command: pipetter.pipette
        sources: $source
        destinationPlate: plate1
        destinations: $$well
        volumes: $$volume
      2:
        command: fluorescenceReader.measurePlate
        object: plate1
        output:
          joinKey: well
```

Since there are two unique `source` values in `data1`, the `data.forEachGroup` command will create two new data tables for it sub-steps:

First table:

```
- {well: A01, volume: 10 ul, source: liquid1}
- {well: A02, volume: 20 ul, source: liquid1}
```

Second table:

```
- {well: B01, volume: 10 ul, source: liquid2}
- {well: B02, volume: 20 ul, source: liquid2}
```

So the sub-steps will be repeated twice, once for new data table. Notice that here we used the single $-prefix for `source`, but the double $$-prefix for `well` and `volume`. Because the values of the well and volume columns are not the same in all rows, they are not automatically added to the scape, and we can't use the single $-prefix.

6.3.4 `data()` directive

The `data()` directive lets you assign a modified version of your data table to a property value. It will be easier to explain this in the section on Substitution, so we'll postpone the discussion till the end of this chapter.

6.4 Scope

Scope is the set of currently active variables in a step. These usually come from one of two sources: 1) the `data` directive and commands, as discussed above, or 2) a loop command like `system.repeat`, which lets you add an index variable to the scope of the sub-steps.

The scope is a kind of stacked-tower structure. When a step pushes variables into scope, they are available to that step's command and all substeps; however, they are not available to sibling or parent steps.

6.5 Substitution

Substitution lets you work with parameters and data tables by inserting their values into the protocol. Roboliq supports three forms: *template* substitution, *scope* substitution, and *directive* substitution.

6.5.1 Scope $ substitution

In scope substition, an expression starting with $ is replaced with a value from the scope. There are various forms of replacement, which we'll dive into now.

`$#...`: pre-scope substitution for parameter values

Parameters are not actually part of the scope, and they are accessible outside of steps as well. This means that they can be used in other parameter values and in object definitions, which is not the case for normal scope variables. You can substitute the value of a parameter named `MYPARAM` by with `$#MYPARAM`.

`${...}`: javascript expression

Roboliq will substitute in the result of a JavaScript expression. The JavaScript expression has access to:

- The scope variables
- `_`: the lodash module and its many functions.
- `math`: the mathjs module and its many functions.

Note that any value JSON value may be returned, whether it's a string, number, boolean, array, or object.

`$(...)`: mathjs calculation

The mathjs module provides a fairly broad range of math operations and is able to handle of units, such as volume. The mathjs expression has access to the current scope variables.

`$...`: scope value substitution

Here you just name the scope variable, and Roboliq will substitute in its value.

If you have activated a data table using the `data` property, then you can use `$colName` to get an array of all the values in the column named `colName`. Furthermore, if any of the data columns are filled with the same value, then that value is added to the scope as `$colName_ONE`, where `colName` is the actual name of the column. For example, if the active data table has a column named `plate` whose entries are all `plate1`, then `$plate1_ONE = "plate1"`.

NOTE: Scope substitution can only be used as a parameter value, but not as a parameter name or part of a longer string. The following uses are invalid:

- `text: "Hello, $name"`: Roboliq only supports scope substitution for an entire value, so the `name` value will not be substituted into this text.
  You can use template substitution for this purpose instead.
- `$myparam: 4`: Roboliq does not support scope substitution for property names. You can use template substitution for this purpose instead.

Examples

Let's look at examples of $-substitutions. Consider this protocol:

```
roboliq: v1
parameters:
  TEXT: { value: "Hello, World" }
objects:
  data1:
    type: Data
    value:
      - {a: 1, b: 1}
      - {a: 1, b: 2}
steps:
  1:
    data: data1
    command: system.echo
    value:
      javascript: "${`${TEXT} ${a} ${__step.command}`}"
      math: "$(a * 10)"
      scopeParameter: $TEXT
      scopeColumn: $b
      scopeOne: $a_ONE
      scopeData: $__data[0].b
      scopeObjects: $__objects.data1.type
      scopeParameters: $__parameters.TEXT.value
      scopeStep: $__step.command
```

The `system.echo` command will output the object described in its `value` parameter. The resulting value is this:

```
javascript: "Hello, World 1 system.echo"
math: 10
scopeParameter: "Hello, World"
```

```
scopeColumn: [1, 2]
scopeOne: 1
scopeData: 1
scopeObjects: "Data"
scopeParameters: "Hello, World"
scopeStep: "system.echo"
```

6.5.2 Template ‘ substitution

Template substitution uses the Handlebars template engine to manipulate text. Template substitution occurs on strings that start and end with a tick (‘). Here’s a simple example that produces a new string:

```
text: "`Hello, {{name}}`"
```

If the `name` in the current scope is “John”, then this will set `text: "Hello, John"`.

If the template substitution result is enclosed by braces or brackets, Roboliq will attempt to parse it as a JSON object. Here’s a trivial example that turns a template substitution into a command, assuming that `name` is currently in scope:

```
1: `{command: "system._echo", text: "Hello, {{name}}"}`
```

6.5.3 Directive () substitution

Directives are substitution functions. The main one is the `data()` directive, which was briefly mentioned above in the Data section. The other directives are also closely related to `Data` objects, and they are discussed more in the chapter on Design Tables.

The `data()` directive lets you assign a modified version of your data table to a property value. The directive can take several properties:

- `where`: same as for the `data` property, this lets you select a subset of rows in the active data table.
- `map`: each row in the data table will be mapped to this value. This is how you can transform your rows.
- `summarize`: like `map`, but for summarizing all the rows into a single row. Summarize has a particularity: all column names are pushed into the current scope as arrays, and they are not overwritten by a single common value even if the column only contains a single value.
- `join`: a string separator that will be used to join all elements of the array (see `Array#join` in some JavaScript documentation).
- `head`: if set to `true`,

Let’s consider some examples using the `data1` table from above. Here is the table:

```
- {well: A01, volume: 10 ul, source: liquid1}
- {well: B01, volume: 10 ul, source: liquid2}
- {well: A02, volume: 20 ul, source: liquid1}
- {well: B02, volume: 20 ul, source: liquid2}
```

and here are the examples:

Directive:
`data(): {where: 'source == "liquid1"'}}`
Result:

```
- {well: "A01", volume: "10 ul", source: "liquid1"}
- {well: "A02", volume: "20 ul", source: "liquid1"}
```

Directive:
`data(): {map: '$volume'}`
Result:

```
["10 ul", "10 ul", "20 ul", "20 ul"]
```

Directive:
`data(): {where: 'source == "liquid1"', map: '$(volume * 2)'}`
Result:

```
["20 ul", "40 ul"]
```

Directive:
`data(): {map: {well: "$well"}}`
Result:

```
- {well: "A01"}
- {well: "B01"}
- {well: "A02"}
- {well: "B02"}
```

Directive:
`data(): {map: "$well", join: ","}`
Result:

```
"A01,B01,A02,B02"
```

Directive:
`data(): {summarize: {totalVolume: '$(sum(volume))'}}`
Result:

```
- {totalVolume: "60 ul"}
```

Directive:
`data(): {groupBy: "source", summarize: {source: '${source[0]}', totalVolume: '$(sum(volume))'}}`
Result:

```
- {source: "liquid1", totalVolume: "30 ul"}
- {source: "liquid2", totalVolume: "30 ul"}
```

Chapter 7

Design Tables

Here we use the term *Design Table* to refer to a data table that is designed for an experiment. It should contain all relevant factors for later analyzing the experimental results.

Experiments on microwell plates can easily involve hundreds of unique liquid combinations, so specifying them manually can be tedious and error-prone. Here we present a short-hand for creating design tables.

WARNING: This is very abstract. Don't bother with it if you're looking for something easy. That said, it makes complex experiments much, much easier to design, trouble-shoot, and analyze.

NOTE: In this chapter, we'll uses the following terms interchangeably: column, factor, variable.

7.1 First examples

Let's create a design table with a single row like this:

plate	source	destination	volume
plate1	water	A01	25 ul

This table could be used to specify a single pipetting operation that dispenses 25 ul of water into well `A01` of `plate1`. We will specify this in Roboliq as follows:

```
roboliq: v1
objects:
  data1:
    type: Data
    description: First example
    design:
      plate: plate1
      source: water
      destination: A01
      volume: 25 ul
---
```

In a Roboliq protocol, designs are placed under the `objects` property. In this case, its name is `data1`, its type is `Data`, and it has a description. The `design` property is where factors are specified.

Let's expand the design table a bit to dispenses 25 ul of water into several destinations.

plate	source	destination	volume
plate1	water	A01	25 ul
plate1	water	B01	25 ul
plate1	water	C01	25 ul

To specify this in Roboliq, we change the `destination` property to a *branching* factor:

```
roboliq: v1
objects:
  data1:
    type: Data
    description: First example
    design:
      plate: plate1
      source: water
      destination*: [A01, B01, C01]
      volume: 25 ul
```

Notice the asterisk in `destination*`. An asterisk at the end of a factor name indicates *branching*. Branching factors require an array of values, and for each value in the array, the existing rows are first replicated and then the value is assigned to the factor in that row.

Now let's assign a different volume to each row:

plate	source	destination	volume
plate1	water	A01	25 ul
plate1	water	B01	50 ul
plate1	water	C01	75 ul

To specify this in Roboliq, we change the `volume` property to an array:

```
roboliq: v1
objects:
  data1:
    type: Data
    description: First example
    design:
      plate: plate1
      source: water
      destination*: [A01, B01, C01]
      volume: [25 ul, 50 ul, 75 ul]
```

When a factor value is an array, a new column is added with those values.

Try it. Copy the above code to a new file in Roboliq's root directory named `data1Test.yaml`. Open a terminal and change directory to the Roboliq root, and run the following command:

```
npm run design -- --path objects.data1 data1Test.yaml
```

It should produce this output:

```
plate   source  destination  volume
======  ======  ===========  ======
plate1  water   A01          25 ul
plate1  water   B01          50 ul
```

```
plate1  water   C01          75 ul
======  ======  ===========  ======
```

Branches can also be specified as integers. If you specify an integer, it creates that many branches which are each given an integer index, as follows:

```
roboliq: v1
objects:
  designInteger:
    type: Data
    design:
      a*: 3
      b*: 3
```

Which creates the following table:

a	b
1	1
1	2
1	3
2	1
2	2
2	3
3	1
3	2
3	3

7.2 Nested branching

Nested branching provides a lot of power to the design specification, but it is also where the complexity starts. Consider this table where two sources are nested in each destination, and each source has its own volume:

plate	destination	source	volume	liquidClass
plate1	A01	water	50 ul	Roboliq_Water_Air_1000
plate1	A01	dye	25 ul	Roboliq_Water_Air_1000
plate1	B01	water	50 ul	Roboliq_Water_Air_1000
plate1	B01	dye	25 ul	Roboliq_Water_Air_1000
plate1	C01	water	50 ul	Roboliq_Water_Air_1000
plate1	C01	dye	25 ul	Roboliq_Water_Air_1000

This can be described in Roboliq as follows:

```
roboliq: v1
objects:
  data2:
    type: Data
    description: Nested example
    design:
      plate: plate1
      destination*: [A01, B01, C01]
      source*:
        water:
```

```
      volume: 50 ul
    dye:
      volume: 25 ul
liquidClass: Roboliq_Water_Air_1000
```

Create a file named `data2Test.yaml` with those contents and run:

```
npm run design -- --path objects.data2 data2Test.yaml
```

Let's walk through this example step-by-step to see how the desired table is achieved.

Step 0: A design starts as a single empty row.

Step 1: `plate: plate1`

This assigns the value `plate1` to the property `plate` in the first row:

plate
plate1

Step 2: `destination*: [A01, B01, C01]`

Three copies of the previous row are created, and a column for `destination` is added to each row, each with its own value:

plate	destination
plate1	A01
plate1	B01
plate1	C01

Step 3: `source*`

This branch has two keys: water and dye. So to start with, two copies are made of each of the previous three rows. The first copy is updated according to the first key, giving us:

plate	destination	source
plate1	A01	water
plate1	B01	water
plate1	C01	water

Then the conditions embedded under `water:` are applied to those rows – in this case, `volume = 50 ul`:

plate	destination	source	volume
plate1	A01	water	50 ul
plate1	B01	water	50 ul
plate1	C01	water	50 ul

For the second table copy, an analogous process sets `source = dye` and `volume = 25 ul`:

plate	destination	source	volume
plate1	A01	dye	25 ul

plate	destination	source	volume
plate1	B01	dye	25 ul
plate1	C01	dye	25 ul

Next those two tables are concatenated, giving us:

plate	destination	source	volume
plate1	A01	water	50 ul
plate1	A01	dye	25 ul
plate1	B01	water	50 ul
plate1	B01	dye	25 ul
plate1	C01	water	50 ul
plate1	C01	dye	25 ul

Step 4: `liquidClass: Roboliq_Water_Air_1000`

Finally, `liquidClass = Roboliq_Water_Air_1000` is assigned to all rows:

plate	destination	source	volume	liquidClass
plate1	A01	water	50 ul	Roboliq_Water_Air_1000
plate1	A01	dye	25 ul	Roboliq_Water_Air_1000
plate1	B01	water	50 ul	Roboliq_Water_Air_1000
plate1	B01	dye	25 ul	Roboliq_Water_Air_1000
plate1	C01	water	50 ul	Roboliq_Water_Air_1000
plate1	C01	dye	25 ul	Roboliq_Water_Air_1000

7.3 Hidden factors

There are generally many ways to achieve the same results. As an example, an alternative way of achieving the same result as above is:

```
roboliq: v1
objects:
  data2:
    type: Data
    description: Nested example
    design:
      plate: plate1
      destination*: [A01, B01, C01]
      .sourceId*:
      - source: water
        volume: 50 ul
      - source: dye
        volume: 25 ul
      liquidClass: Roboliq_Water_Air_1000
```

In this case, the branching factor is `.sourceId*` and it's an array. The period prefix hides that column, and the final results are the same as above.

7.4 Actions

Roboliq provides various designs "actions" that can be used for more sophisticated values. The most important ones are:

- allocateWells
- range
- calculate
- case

7.4.1 `allocateWells`

Let's take a look at an example:

```
replicate*: 2
well=allocateWells:
  rows: 8
  columns: 12
```

An action is indicated with the "="-infix. So in the case of `well=allocateWells`, the factor name is `well`, the action is `allocateWells`, and the properties are the arguments to the action. In this case, `rows` and `columns` tells the plate dimension we want to get wells for, and 2 wells will be allocated since the table has two rows:

```
{replicate: 1, well: A01}
{replicate: 2, well: B01}
```

7.4.2 `range`

The `range` action gives you an integer sequence. It accepts these arguments:

- `from`: the integer to start at (optional)
- `till`: the integer to end at (optional)
- `step`: the distance between generated integers (default = 1)

Here's an example:

```
a*: 2
b*: 2
c=range: {}
d=range: {from: 10, step: 10}
```

Which produces this result:

a	b	c	d
1	1	1	10
1	2	2	20
2	1	3	30
2	2	4	40

The first range, `c`, just numbers all the rows starting with 1. The second range, `d`, starts numbering at 10 and procedes in steps of 10.

7.4.3 calculate

The `calculate` action takes a string to be parsed by mathjs. The calculation will be made for each row individually. Here's an example:

```
a*: 3
volume=calculate: '(a * 10) ul'
more=calculate: '(50 ul) - volume'
```

And here is the result:

a	volume	more
1	10 ul	40 ul
2	20 ul	30 ul
3	30 ul	20 ul

Alternatively, the `calculate` action can accept parameters:

- `value`: the string to parse
- `units`: the units of the final output

```
a*: 3
volume=calculate:
  value: '(a * 10)'
  units: ul
```

With this output:

a	volume
1	10 ul
2	20 ul
3	30 ul

7.4.4 case

A `case` action takes an array of cases and tests them against each row of the table. The first case whose `where` statement is missing or evaluates to `true` will be applied. The individual case items take these arguments:

- `where` - an optional mathjs statement that will be evaluated on each row
- `design` - a design specification that will be applied to the matching rows

Here's an example:

```
a*: 3
volumeCase=case:
  - where: a < 2
    design:
      volume: 10 ul
  - design:
      volume: 12354 ul
```

a	volumeCase	volume
1	1	10 ul
2	2	12345 ul

a	volumeCase	volume
3	2	12345 ul

7.5 Step and data nesting

You can only load one design per step, but you can nest steps and load another design in the sub-step. Consider these two excerpts of designs:

```
data1:
  design:
    a: Alice
    b: *3
    c: Charles
    d: Daniel

data2:
  design:
    d: David
```

Let's use them in these steps:

```
1:
  data: {source: data1}
  description: "`{{a}} {{c}} {{d}}`"
  1:
    data: {source: data2}
    description: "`{{a}} {{c}} {{d}}`"
```

The descriptions should be expanded as follows:

1.description: "Alice Charles Daniel" 1.1.description: "Alice Charles David"

In 1.1, "$$b" does not exist, but "$a" and "$c" still do. That is to say: column data from a previous `data` directive are not carried into sub-steps with a new `data` directive, but the values that were the same for all columns remain in scope.