# Automated planning enables complex protocols on liquid-handling robots

# —Supporting Information—

Ellis Whitehead, Fabian Rudolf, Hans-Michael Kaltenbach, and Jörg Stelling*

*Department of Biosystems Science and Engineering, ETH Zurich and SIB Swiss Institute of Bioinformatics, Mattenstrasse 26, 4058 Basel, Switzerland*

E-mail: joerg.stelling@bsse.ethz.ch

More detailed documentation of `Roboliq` is available online:

- Source code:

  `https://git.bsse.ethz.ch/csb/roboliq`

- User manual:

  `http://intra.csb.ethz.ch/roboliq/docs/manual`

- `Roboliq` command reference:

  `http://intra.csb.ethz.ch/roboliq/docs/protocol/commands.html`

- Evoware command reference:

  `http://intra.csb.ethz.ch/roboliq/docs/protocol/evowareCommands.html`

- `Roboliq` code for proof-of-principle applications:

  `https://git.bsse.ethz.ch/csb/roboliq/tree/master/examples`

## Optimization and automated parameter selection

`Roboliq` implements optimizations for labware transfers and pipetting procedures. Some of these optimizations assume that the configuration file is well constructed, which is not an easy task. Once such as configuration file is created, however, the optimizations usually produce sensible defaults so that the user does not need to specify the transfer or pipetting parameters manually.

Regarding labware transfers, the robot configuration file defines all valid single-step transfers. A "single-step transfer" consists of three pieces of data: the robot arm, the movement method, and the pair of bench locations to move between. In order to move labware between two arbitrary bench locations, `Roboliq` iteratively explores possible solutions. Specifically, it first searches for a single-step solution; if none is feasible, two-step solutions are tested, and so forth. Through this breadth-first search, feasible transfers of labware will always involve the minimum number of steps. For our setup, all locations can be reached in three or fewer steps, and `Roboliq` raises an error if it cannot find a solution with three steps. The maximum number of steps can be set in the robot configuration file.

`Roboliq` optimizes pipetting procedures in three ways. First of all, `Roboliq` allows the user to specify sterility requirements. For example, we can specify that the tips need to be decontaminated before aspirating from a source that contains cells. One can also specify that no rinsing is required between transfers of water aliquots to multiple wells, if the tips did not touch any other liquids because all dispenses took place above the well. Secondly, `Roboliq` supports configurable pipetting rules (which the user can override whenever necessary). The rules can select tip size and other pipetting parameters based on well volume, aliquot volume, and liquid contents. For example, the rules can specify different tip sizes for different aliquot volumes and different dispense heights depending on whether the destination well is empty or full. Finally, `Roboliq` takes advantage of multi-tip pipetters by performing as many aspirations and dispenses in parallel as possible.

**Table S1:** Comparison of software systems for automation. Each row of the table represents an important feature for general portable protocols, and the columns are some of the important software systems discussed in the Introduction. Checkmarks indicate that a system supports the given feature, and a tilde indicates partial support. The *Antha* column has question marks because the feature support is still unclear. *General Experiment Framework*: the system can be used for general experimental tasks, as opposed to being focused on a single type of protocol. *Portable Protocols*: the system supports protocols that can be used in different labs. *Low-Level Optimization*: the system allows users to tweak the low-level details of a protocol to optimize execution on a specific platform. *Executable Framework*: the system provides a means to execute its protocols. *Software Interface*: the system is designed to interface smoothly with third-party software. *Adaptable / Extensible*: the system can be adapted to new labs and its command set can be extended in a portable manner. Some automation systems are concerned with a single task, like j5 for optimizing DNA assembly. Others, like BioCoder and EXACT2, provide formal protocol specifications, but they do not have an actual execution platform. Recent developments, like Antha, appear promising, but their capabilities are not yet clear. PR-PR can be considered the reference platform for portable programming of liquid handling robots. However, it has several short-comings that we wanted to address: the protocol format is not really portable or flexible, it's design hinders interfacing with other software, and it is difficult to adapt and extend. `Roboliq` is our system, which we designed as a general experiment framework that can handle portable protocols, but it is also flexible enough to allow low-level optimization. It can execute the experiments, interface with third-party software, and is designed for adaptation and extension. `Roboliq` 's most significant difference to Autoprotocol is that it generates low-level, executable code, whereas Autoprotocol is a formal language specifying a high-level protocol. In addition, (i) `Roboliq` allows more flexible parameter overriding for potential portability (Autoprotocol requires all specified parameters to be in the protocol itself), (ii) it facilitates more low-level control, (iii) it allows for more powerful commands through expansion of commands and loops, (iv) `Roboliq` supports more advanced variable substitution and tidy data output, and (v) Autoprotocol does not provide any support for AI (automated planning), so all that would need to go into the backend compiler, whereas `Roboliq` 's main compiler contains the AI so that the backend can be as simple as possible.

| | j5 | BioCoder | EXACT2 | Autoprotocol | Antha | PR-PR |
|---|---|---|---|---|---|---|
| General Experiment Framework | | *X* | *X* | *X* | *X* | *X* |
| Portable Protocols | *X* | *X* | *X* | *X* | ? | ~ |
| Low-Level Optimization | | | | *X* | ? | ~ |
| Executable Framework | *X* | | | ~ | ? | *X* |
| Software Interface | *X* | | | *X* | ? | |
| Adaptable / Extensible | | | | *X* | ? | ~ |

**Table S2:** List of `Roboliq`'s standard commands. For more details, see the the online documentation at `https://git.bsse.ethz.ch/csb/roboliq/protocol/commands.html`.

| Command | Short description |
|---|---|
| **absorbanceReader** | |
| `measurePlate` | Measure the absorbance of a plate. |
| **centrifuge** | |
| `centrifuge2` | Centrifuge two plates. |
| `insertPlates2` | Insert up to two plates into the centrifuge. |
| **data** | |
| `forEachGroup` | Perform sub-steps for every grouping of rows in the active data table. |
| `forEachRow` | Perform sub-steps for every row in the active data table. |
| **equipment** | |
| `_run` | Run the given equipment. |
| `close` | Close the given equipment. |
| `open` | Open the given equipment. |
| `openSite` | Open an equipment site. |
| `start` | Start the given equipment. |
| `stop` | Stop the given equipment. |
| **fluorescenceReader** | |
| `measurePlate` | Measure the fluorescence of wells on a plate. |
| **incubator** | |
| `incubatePlates` | Incubate the given plates. |
| `insertPlates` | Insert up to two plates into the incubator. |
| `run` | Run the incubator with the given program |
| **pipetter** | |
| `_aspirate` | Aspirate liquids from sources into syringes. |
| `_dispense` | Dispense liquids from sryinges into destinations. |
| `_measureVolume` | Measure well volume using pipetter tips. |
| `_mix` | Mix liquids by aspirating and re-dispensing. |
| `_pipette` | Pipette liquids from sources to destinations. |
| `_punctureSeal` | Puncture the seal on a plate using pipetter tips. |
| `_washTips` | Clean the pipetter tips by washing. |
| `cleanTips` | Clean the pipetter tips. |
| `measureVolume` | Measure well volume using pipetter tips. |
| `mix` | Mix well contents by aspirating and re-dispensing. |
| `pipette` | Pipette liquids from sources to destinations. |
| `pipetteDilutionSeries` | Pipette a dilution series. |
| `pipetteMixtures` | Pipette the given mixtures into the given destinations. |
| `punctureSeal` | Puncture the seal on a plate using pipetter tips. |
| **scale** | |
| `weigh` | Weigh an object. |
| **sealer** | |
| `sealPlate` | Seal a plate. |

| Command | Short description |
|---|---|
| **shaker** | |
| `run` | Run the shaker. |
| `shakePlate` | Shake a plate. |
| | |
| **system** | |
| `_description` | Include the value as a description in the generated script. |
| `_echo` | Include the value in the generated script for trouble-shooting. |
| `call` | Call a template function. |
| `description` | Include the value as a description in the generated script. |
| `echo` | Include the value in the generated script for trouble-shooting. |
| `if` | Conditionally execute steps depending on a conditional test. |
| `repeat` | Repeat sub-steps a given number of times. |
| `runtimeExitLoop` | Test at run-time whether to exit the current loop. |
| `runtimeLoadVariables` | Load the runtime values into variables. |
| `runtimeSteps` | Handle steps that require runtime variables. |
| | |
| **timer** | |
| `_sleep` | Sleep for a given duration using a specific timer. |
| `_start` | Start the given timer. |
| `_stop` | Stop the given timer. |
| `_wait` | Wait until the given timer has reacher the given elapsed time. |
| `doAndWait` | Start a timer, perform sub-steps, then wait till duration has elapsed. |
| `sleep` | Sleep for a given duration. |
| `start` | Start a timer. |
| `stop` | Stop a running a timer. |
| `wait` | Wait until the given timer has reacher the given elapsed time. |
| | |
| **transporter** | |
| `_moveLidFromContainerToSite` | Transport a lid from a container to a destination site. |
| `_moveLidFromSiteToContainer` | Transport a lid from an origin site to a labware container. |
| `_movePlate` | Transport a plate to a destination. |
| `doThenRestoreLocation` | Perform steps, then return the given labwares to their prior locations. |
| `moveLidFromContainerToSite` | Transport a lid from a container to a destination site. |
| `moveLidFromSiteToContainer` | Transport a lid from an origin site to a labware container. |
| `movePlate` | Transport a plate to a destination. |

**Table S3:** List of low-level commands for Tecan Evoware. For more details, see the the online documentation at `https://git.bsse.ethz.ch/csb/roboliq/protocol/evowareCommands.html`.

| Command | Short description |
|---|---|
| **evoware** | |
| `_execute` | An Evoware Execute command |
| `_facts` | An Evoware FACTS command |
| `_raw` | An Evoware direct command |
| `_subroutine` | An Evoware 'Subroutine' command |
| `_userPrompt` | An Evoware UserPrompt command |
| `_variable` | Set an Evoware variable |

**Table S4:** Extended protocol structure. In order to support programming, `Roboliq` extends the protocol structure with these fields that may contain JavaScript code. For more details, see the online documentation at `https://git.bsse.ethz.ch/csb/roboliq/manual/configuration.html`.

| Field | Description |
|---|---|
| `predicates` | an array of logical predicates used by the Automated Planning algorithm. We use Warren Sack's JSON implementation for encoding logic[1] in combination with his implementation of the SHOP2 algorithm for automated planning[2]. |
| `objectToPredicateConverters` | a map from an object type to a function that produces predicates to describe an object of that type for the Automated Planning algorithms. |
| `commandHandlers` | a map from a command name to a function that handles a command for a protocol step. |
| `planHandlers` | a map from the name of a logical action to a function that outputs the `Roboliq` command for that action. |

```
mergeObjects(o1, o2):
  result = empty object
  keys = union of keys in o1 and o2
  for each key:
    if o1[key] and o2[key] are in both objects:
      result[key] = mergeObjects(o1[key], o2[key])
    else if o2 has key:
      result[key] = o2[key]
    else:
      result[key] = o1[key]

mergeProtocols(p1, p2):
  result = mergeObjects(p1, p2)
  result['predicates'] =
    concatenate 'predicates' from p1 and p2
  result['taskPredicates'] =
    concatenate 'taskPredicates' from p1 and p2

loadProtocol(url, params):
  protocol = load url as JSON, YAML, or JavaScript with params
  if protocol has 'requires' key:
    module = empty object
    for each requirement in protocol.requires:
      protocol2 = loadProtocol(requirement url, requirement params)
      module = mergeObjects(module, protocol2)
    protocol = mergeObjects(module, protocol)
    remove 'requires' key from protocol
  return protocol
```

**Figure S1:** Pseudocode for merging objects, merging protocols, and loading protocols. JSON data consists of several types of values: basic values such as numbers and strings, arrays, and objects. A JSON object is a collection of key/value pairs. `mergeObjects` inspects two objects, whereby the fields of the second object have priority – if they both have a particular key whose values are also objects, those values are recursively merged; otherwise if the second object has the key, take its value; otherwise take the value from the first object. `mergeProtocols` differs from `mergeObjects` merely by concatenating the arrays for `predicates` and `taskPredicates`, rather than just taking the value from `p2` if available. `loadProtocol` loads the given URL as a JSON object, a YAML object, or a JavaScript function (to which it passes extra parameters if supplied). If the resulting JavaScript object has a `requires` key, it will recursively load the required modules and merge them.

```
expandProtocol(protocol):
  objects = clone a copy of protocol.objects
  expandStep(protocol, "", objects);

expandStep(protocol, id, objects):
  step = lookup step with id in protocol
  if step has 'command' key:
    predicates = protocol.predicates ++ objectPredicates(objects)
    handler = protocol.commandHandlers[step.command]
    result = handler(step, objects, predicates,
                     protocol.planHandlers)
    protocol.cache[id] = result
    protocol.errors[id] = result.errors
    abort if there were errors
    if result has 'expansion' key:
      merge result.expansion into step (mutates protocol too)
    protocol.effects[id] = result.effects
    for effect in result.effects:
      merge effect into objects
  for each substep in step:
    substepId = id + '.' + substep index
    expandStep(protocol, substepId, objects)
```

**Figure S2:** Pseudocode for expanding the steps of a protocol. `expandProtocol()` starts the expansion process by cloning a mutable copy of the protocol's objects and calling `expandStep()`. In `expandStep()`, we first check whether the current step contains a command. If so, the original predicates are merged with the dynamic object predicates, the command handler is invoked, its results are stored, and errors, expansions, and effects are handled. Finally, if the step has sub-steps, each of them is expanded in turn.

```
objects:
  balancePlate:
    type: Plate
    description: balance plate for centrifuge
    model!: ourlab.model.plateModel_384_square
    location!: ourlab.mario.site.P4
  mixPlate:
    type: Plate
    model!: ourlab.model.plateModel_384_square
    location!: ourlab.mario.site.P3

  tubes1!:
    type: Plate
    description: GFP eppendorf tubes
    model: ourlab.model.tubeHolderModel_1500ul
    location: ourlab.mario.site.T3
  trough1!:
    type: Plate
    description: trough for water/glycerol/salt mix
    model: ourlab.model.troughModel_100ml
    location: ourlab.mario.site.R6
    contents: [Infinity l, saltwater]
  sourcePlate1!:
    type: Plate
    description: buffer plate
    model: ourlab.model.plateModel_96_dwp
    location: ourlab.mario.site.P2
```

**Figure S3:** pH experiment specification: Labware. This protocol excerpt defines the labware used in the pH experiment. The `!`-suffix indicates lab-specific values that were set to run the experiment on our robot. Each labware has a `type`, `description`, `model`, and `location`. Note that `Roboliq` does not have a separate type for troughs, so the troughs also have type 'Plate'. The `model` and `location` values are unique identifiers defined in configuration file for the available labware models and bench locations. `through1` has an additional property `contents` that specifies its initial liquid contents; this is an array whose first element is the volume and second element is the liquid. In this case, the volume is given as `Infinity l`, but an exact value could be given instead.

```
  saltwater:    {type: Liquid, group: Buffers, wells!: trough1(C01 down to F01)}
  hepes_850:    {type: Liquid, group: Buffers, wells!: sourcePlate1(A01 down to D01)}
  hepes_650:    {type: Liquid, group: Buffers, wells!: sourcePlate1(A02 down to D02)}
  pipes_775:    {type: Liquid, group: Buffers, wells!: sourcePlate1(A03 down to D03)}
  pipes_575:    {type: Liquid, group: Buffers, wells!: sourcePlate1(A04 down to D04)}
  mes_710:      {type: Liquid, group: Buffers, wells!: sourcePlate1(A05 down to D05)}
  mes_510:      {type: Liquid, group: Buffers, wells!: sourcePlate1(A06 down to D06)}
  acetate_575:  {type: Liquid, group: Buffers, wells!: sourcePlate1(A07 down to D07)}
  acetate_375:  {type: Liquid, group: Buffers, wells!: sourcePlate1(A08 down to D08)}

  sfGFP:        {type: Liquid, group: GFPs, description: wild type, wells!: tubes1(A01)}
```

**Figure S4:** pH experiment specification: Liquids. This defines the ten liquids used in this paper: salt water, buffers, and protein. They each have a `type` of `Liquid`, an optional `group` to help organization then, and a `wells` property that specifies where the liquid sources should be. The `wells` property has a `!`-suffix to indicate that it is lab-specific, because another lab could easily choose to put the liquids somewhere else.

```
mixtures:
  type: Variable
  calculate:
    "#createPipetteMixtureList":
      replicates: 3
      items:
      - source: saltwater
        volume: 40ul
      - "#gradient":
        - {source1: acetate_375, source2: acetate_575, volume: 30ul, count: 8, decimals: 1}
        - {source1: mes_510,     source2: mes_710,     volume: 30ul, count: 7, decimals: 1}
        - {source1: pipes_575,   source2: pipes_775,   volume: 30ul, count: 5, decimals: 1}
        - {source1: hepes_650,   source2: hepes_850,   volume: 30ul, count: 5, decimals: 1}
      - source: sfGFP
        volume: 5ul
        clean: thorough
        cleanBetweenSameSource: flush
        program!: Roboliq_Water_Wet_1000_mix3x50ul

mixtureWells:
  type: Variable
  calculate:
    "#createWellAssignments":
      list: mixtures
      wells: mixPlate(all row-jump(1))
```

**Figure S5:** pH experiment specification: Mixtures. Here we define two variables. `mixtures` is a mixture matrix that is calculated by the `createPipetteMixtureList()` function. It specifies three replicates per combination, where each mixture has 40 µL of salt water, 30 µL of one of four buffer systems, and 5 µL of sfGFP. For the sfGFP component, further pipetting parameters are included to guide cleaning and mixing. The `mixtureWells` variable assigns the wells that will be used for mixing.

```
steps:
  1:
    description: Prepare the mixture plate with a range of pH levels
    1:
      command: pipetter.pipetteMixtures
      mixtures: mixtures
      destinations: mixtureWells
      clean: flush
      cleanBegin: thorough
      cleanBetweenSameSource: none
      cleanEnd: thorough

    2:
      command: sealer.sealPlate
      object: mixPlate

    3:
      command: fluorescenceReader.measurePlate
      object: mixPlate
      program:
        excitation: 488nm
        emission: 510nm
      programFile: ./ph.mdfx
```

**Figure S6:** pH experiment specification: Steps. This is an excerpt of the step definitions that doesn't include the loop for repeated measurements. The first step pipettes the mixtures, the second step seals the plate, and the third step measures absorbance (it uses the user-defined file 'ph.mdfx' as a template for the measurements).

# References

1. Sack, W. A JavaScript-based HTN Planner. 2010; `http://danm.ucsc.edu/~wsack/Plan/abstract.html`.

2. Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003) SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research 20*, 379–404.