Deliverable D3.5

Funding Scheme: THEME [ICT-2007.8.0] [FET Open]

Paving the Way for Future Emerging DNA-based Technologies: Computer-Aided Design and Manufacturing of DNA libraries

Grant Agreement number: 265505

Project acronym: CADMAD

Deliverable number: D3.5

Deliverable name: RoboEase source code

Contractual Date[1] of Delivery to the CEC: M24 (not specified in Annex I)
Actual Date of Delivery to the CEC: M24
Author(s)[2]: Ellis Whitehead, Joerg Stelling
Participant(s)[3]: ETHZ
Work Package: WP3
Security[4]: Int
Nature[5]: R
Version[6]: 0.0
Total number of pages:

1 As specified in Annex I

2 i.e. name of the person(s) responsible for the preparation of the document

3 Short name of partner(s) responsible for the deliverable

4 The Technical Annex of the project provides a list of deliverables to be submitted, with the following classification level:
Pub - Public document; No restrictions on access; may be given freely to any interested party or published openly on the web, provided the author and source are mentioned and the content is not altered.
Rest - Restricted circulation list (including Commission Project Officer). This circulation list will be designated in agreement with the source project. May not be given to persons or bodies not listed.
Int - Internal circulation within project (and Commission Project Officer). The deliverable cannot be disclosed to any third party outside the project.

5 **R (Report):** the deliverables consists in a document reporting the results of interest.
P (Prototype): the deliverable is actually consisting in a physical prototype**,** whose location and functionalities are described in the submitted document (however, the actual deliverable must be available for inspection and/or audit in the indicated place)
D (Demonstrator): the deliverable is a software program, a device or a physical set-up aimed to demonstrate a concept and described in the submitted document (however, the actual deliverable must be available for inspection and/or audit in the indicated place)
O (**Other):** the deliverable described in the submitted document can not be classified as one of the above (e.g. specification, tools, tests, etc.)

6 Two digits separated by a dot:
The first digit is 0 for draft, 1 for project approved document, 2 or more for further revisions (e.g. in case of non acceptance by the Commission) requiring explicit approval by the project itself;
The second digit is a number indicating minor changes to the document not requiring an explicit approval by the project.

Abstract

We have created a software system for high-level control of a liquid handling robot. Details of the implementation are described, and we consider the results with respect to abstraction, flexibility, feedback handling, expressiveness, intelligence, and portability. The source code of the implementation is available internally and it will be made open-source in a procedure to be determined by the project partners.

Keywords[7]**:**

Liquid handling robot, RoboEase, programming language

1. Introduction

Aim / Objectives

The aim of the work in T3.3 was the development of a high-level programming language for liquid-handling robots. Specifically, the language was to be designed to possess the following properties.

- *High-level*: Most tasks we want to perform in the laboratory (e.g., for DNA library construction) involve multiple steps, such that in addition to low-level commands, we want to be able to have high-level commands (such as "do PCR") which may compile down to many low-level commands.
- *Flexible*: The system should support a wide range of tasks, for instance, a range of different protocols in molecular biology and biochemistry.
- *Feedback handling*: we would like to support fully automated feedback loops, whereby the next step is determined by measurements performed in the previous steps to enable error correction in real-time.
- *Expressive*: the language should be expressive enough to let the users avoid repeating themselves too much by supporting language constructs such as variables and loops.
- *Intelligent*: the commands should handle as much complexity as they can, so that the user does not need to, for instance, specify all detailed parameters in a high-level script. This also means that the system should perform validation of the scripts to make sure that the commands can be plausibly executed.
- *Portable*: it should be possible to take a script from one lab and execute it in another lab with a different robot.

State of the Art

None of the existing languages for programming liquid-handling robots meets all (or even a majority) of the specification criteria listed above for the following reasons:

- *Not high-level*: Few languages offer support for multi-step commands in a convenient manner (the original RoboEase developed at Weizmann is an exception).
- *Not flexible*: Few languages offer much parameterization of commands or they only have limited support for subroutines. For example, Tecan's subroutines do not allow for a dynamic choice of which wells to pipette from.
- *Limited feedback handling*: Feedback handling tends to be limited to very simple cases that only have a few possible responses. For more complex cases, the robot operator will likely need to generate a new script and manually start it, even if the decision about what to do next is a simple deterministic matter.
- *Not expressive*: User-defined variables and subroutines are very limited in their form and type for all previous languages.
- *Not especially intelligent*: Although most languages perform a certain amount of validation, few are able to relieve the user of specifying parameters and tasks which are obvious when the context is considered.

[7] Keywords that would serve as search label for information retrieval

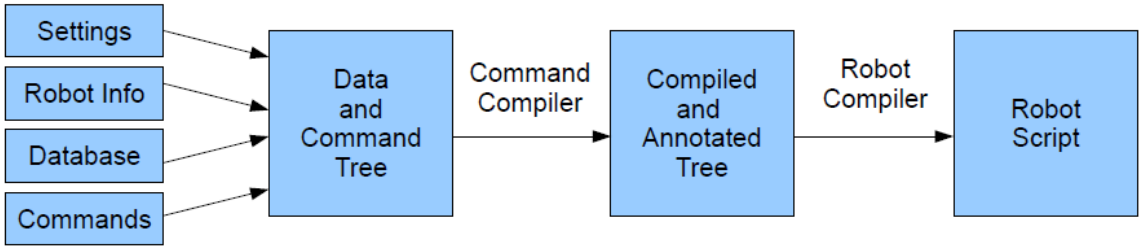
- *Not portable*: Scripts written for a particular robot setup cannot be run on another robot without extensive adaptation.

Innovation

Our system manages to gracefully combine low-level and high-level commands, making both types available to the user. The system also implements a uniform format for data exchange, so that data can be just as easily extracted from a database as being supplied by the user. Furthermore, robot state information is tracked in detail and made available to each command. This has two positive consequences. First, the high-level commands are able to make many more intelligent decisions without the user needing to specify the details; for example, the command to run the thermal cycler on a plate can automatically transfer the plate to the thermal cycler. Second, since fewer parameters and low-level commands need to be specified by the user, the scripts are more easily ported between labs.

2. Implementation

The components of the system are illustrated in the following diagram.



On the left side, we have the input to the system: general settings supplied by the user, configuration information about the target robot, a database supplying information about the substances and plates used in experiments, and a list of commands for execution on the robot. The information needed to execute the commands is pooled together and represented as a complete data tree. The data tree is then passed to the command compiler, producing a new tree of low-level commands and information useful for troubleshooting. Finally, the low-level commands are transformed into a format that can be executed on the target robot.

3. Results

Examples of input and output

Definitions of a substance and a plate are shown here in YAML format. (YAML is a plain text format for storing objects in a programming language. It is similar to XML, but much more concise. A link to more information can be found in the References section.) The substance named SEQUENCE_01 is a type of DNA with the given sequence. The plate E2215 is defined with a specific plate model and barcode.

```
substances:
  SEQUENCE_01: !dna
    sequence: TATAACGTTACTGGTTTCATGAATTCTTGTTAATTCAGTAAATTTTC

plates:
  E2215:
    model: D-BSSE 96 Well Costar
    barcode: 059662E2215
```

The next diagram shows examples of a plate model definition, the specification of a pipette device driver, and the specification of two command handlers for aspiration and dispense.

```
plateModels:
  D-BSSE 96 Well PCR Plate: { rows: 8, cols: 12, volume: 200 ul }

devices:
- !!roboliq.labs.bsse.PipetteDevice

commandHandlers:
- !!roboliq.commands.pipette.AspirateCmdHandler
- !!roboliq.commands.pipette.DispenseCmdHandler
```

The following scheme demonstrates a simple “pipette” command excerpted from a larger command list. It transfers 5µl of liquid from the plate with ID “P1” and well A01 to the plate with ID “P4”, well C03.

```
- !pipette
  src: P1(A01)
  dest: P4(C03)
  volume: 5 ul
```

Once the above command has been passed through the command compiler, an expanded and annotated tree is produced. The annotation command with automatic documentation and a list of events can be seen here:

```
output:
- command: !pipette
    src: P1(A01)
    dest: P4(C03)
    volume: 5 ul
  doc: pipette 5ul of water from P1(A01) to P4(C03)
  events:
  - P1(A01): !rem {volume: 5e-6}
  - P4(C03): !add {src: P1(A01), volume: 5e-6}
```

Further down in the annotated tree we find the low-level commands which will be translated by the robot compiler. Here for example we see the low-level “aspirate” command, where a specific tip and liquid handling policy have been chosen.

```
...
  children:
  - command: !aspirate
      items:
      - tip: TIP1
        well: P1(A01)
        volume: 5e-6
        policy: Roboliq_Water_Dry_1000
...
```

Core classes in source code

At the core of the language is a library written in a Java-compatible language called Scala. The main objects which the library utilizes and manipulates are substances, liquids, vessel contents, vessels, plates, and labware models. The corresponding classes can be found in the source package “roboliq.core” and they are described next.

Substances: We have two primary categories of substances: liquids and powders. A liquid has a volume and can be pipetted (see SubstanceLiquid). Powders are specified in mol units. Currently there are two powder subclasses: SubstanceDna and SubstanceOther.

Liquids: The Liquid represents one or more solvents containing zero or more solutes. A Liquid represents the ratios of its contents and is therefore independent of volume. Any mixture with the same ratios is considered to be the same liquid, so a particular liquid can be present in multiple vessels. However, the class does not interface well with the other classes; it is currently being redesigned to properly represent ratios in a manner similar to VesselContent, and will be used in the next release.

Vessel contents: The vessel's contents are represented by VesselContent. This is similar to the description of liquids above, except that absolute amounts are used instead of ratios, and the contents are specific to a particular vessel.

Vessels: A vessel is an object which can contain substances, or more precisely has VesselContent. There are two kinds of vessels: PlateWell and Tube. Also of interest is the poorly-named Well2, which represents a vessel on a vessel holder. A PlateWell is automatically also a Well2, whereas a Tube doesn't have Well2 information until it has been placed on a rack.

Vessel holders: Conceptually, there are two kinds of holders: 1) plates, which have wells built into them, and 2) racks, which can accommodate removable tubes. Currently, we assume that the tubes on a rack will not change during the execution of a protocol, so we use Plate for both cases.

Labware models: Every piece of labware is considered to be an instance of a labware model, as follows:

- A Tip is an instance of a TipModel
- A Plate is an instance of a PlateModel
- A Tube is an instance of a TubeModel

State and events: “State” refers to the properties of an object which can change over time, and an object's state information represents the cumulative effect of events. StateQuery and StateMap are interfaces to the state of all objects in the system. There are two concrete implementations, an immutable RobotState and a mutable StateBuilder. Events have an update method to update the object's state.

Databases: The term "database" is used in a broad sense here to mean a large set of data which can be queried by ID. There are two "databases" in roboliq.

- BeanBase: holds the YAML JavaBeans which get read in from files.
- ObjBase: holds instantiations of the objects required for executing a protocol, and also holds a map of the initial state of those objects.

Commands: Command data is contained in a CmdBean. The code which actually handles the command processing is in a class which inherits from CmdHandler.

Command evaluation

There are two phases to the evaluation of a command. The *check phase* gathers all variables which will be needed for its execution from the ObjBase. A command may need to obtain several kinds of information: objects, object states, object settings. The *handle phase* translates the command into a list of subcommands or tokens. Tokens are used by the robot-specific translator to generate its scripts.

When a command is checked, it may find that 1) not all information is available which it needs or 2) some preprocessing needs to be performed.

Missing information includes things like the location where a plate should be placed on the bench, which new plates to use when new plates are required, or which of several thermocyclers to use if more than one is available. After getting a list of missing information, the processors (Processor) can try to find sensible defaults. The remaining values must be chosen by the user. Once that is done, the commands can be processed again, now with the complete information set.

Levels of command token abstraction

Command trees go through several levels of processing, starting at the most abstract level L4 and progressing down to concrete low-level robot instructions at L0.

- L4: not all parameters need to be specified if defaults can be chosen.

- L3: tokens have all L4 parameters specified.

Translation L3 to L2: here is where the bulk of decision making can be performed. All well locations need to be made explicit rather than referring to liquids or all wells of a plate.

- L2: tokens have access to extended configuration and state information. The tokens at this level are more concise than the L1 commands, but they are state-dependent.

- L1: tokens do not have access to RobotState information. The idea is to have the command fully specified by its parameters. They should be as simple as possible in order to make the cross-platform translators for various robot platforms as simple as possible.

Translation L1 to L0: performed by a translator that was designed for a specific robot platform.

- L0: concrete tokens for the target robot.

4. Conclusions

The target audience for our robot control system differs from that of the original RoboEase language developed at the Weizmann Institute. That language was intended to enable biologists to create scripts for implementing protocols on the robot. In contrast, the system developed by ETHZ is primarily intended to be used as a tool by higher-level software or experienced programmers, while still permitting simple scripting by less sophisticated users. This change of direction came about due to the need for producing complex, flexible scripts, which stands in opposition to the need to have a simple language for biological users. After observing that most scripts written by biologists only contained a single command, we concluded that a graphical user interface could better serve their use case, and by allowing for more complexity in the language itself, a much larger range of applications could be addressed.

The resulting system for generating scripts is quite flexible without being steeped in complexity. It can be either controlled via plain text files or via direct library calls, and it can cull the data need for commands (such as available plates and substances) from the protocol tree or from a database. Referring back to the properties we listed in the introduction, the system is high-level, flexible, expressive, and intelligent. The capacity for handling feedback is ongoing work (not originally planned for in Annex I, but an additional requirement for quality control and automatic error-correction methods), but we have not achieved results better than other systems yet. And lastly, portability has significantly improved upon beyond previous languages, as a natural result of fewer parameters and low-level commands needing to be specified by the user, but more work may be able to provide further advances.

5. References

- Scala: http://www.scala-lang.org/
- YAML: http://www.yaml.org/
- A secure location for the source code needs to be determined. This will be established before the review.

6. Abbreviations

List all abbreviations used in the document arranged alphabetically.

DNA	Deoxyribon Nucleic Acid
PCR	Polymerase Chain Reaction
YML	Yet Another Multicolumn Layout
XML	Extensible Markup Language