# Building an Open Representation for Biological Protocols

BRYAN BARTLEY, JACOB BEAL, and MILES ROGERS, Raytheon BBN Technologies, USA
DANIEL BRYCE and ROBERT P. GOLDMAN, SIFT, LLC, USA
BENJAMIN KELLER, University of Washington, USA
PETER LEE, Ginkgo Bioworks, USA
VANESSA BIGGERS and JOSHUA NOWAK, Strateos, Inc., USA
MARK WESTON, Netrias, Inc., USA

Laboratory protocols are critical to biological research and development, yet difficult to communicate and reproduce across projects, investigators, and organizations. While many attempts have been made to address this challenge, there is currently no available protocol representation that is simultaneously unambiguous enough for precise interpretation and automation, yet simultaneously abstract enough to enable reuse and adaptation. The Protocol Activity Markup Language (PAML) is a free and open protocol representation aiming to address this gap, building on a foundation of UML, Autoprotocol, and SBOL RDF. PAML provides a representation both for protocols and for records of their execution and the resulting data, as well as a framework for exporting from PAML for execution by either humans or laboratory automation. PAML is currently implemented in the form of an RDF knowledge representation, specification document, and Python library, can currently be exported for execution as either a manual "paper protocol" or Autoprotocol, and is being further developed as an open community effort.

Additional Key Words and Phrases: protocol, biology, representation, UML, RDF, SBOL

**ACM Reference Format:**
Bryan Bartley, Jacob Beal, Miles Rogers, Daniel Bryce, Robert P. Goldman, Benjamin Keller, Peter Lee, Vanessa Biggers, Joshua Nowak, and Mark Weston. 2022. Building an Open Representation for Biological Protocols. 1, 1 (November 2022), 21 pages. https://doi.org/TBD

## 1 INTRODUCTION

Laboratory protocols are critical to biological research and development. Protocols are often difficult to communicate or reproduce, however, given the differences in context, skills, and resources between different projects, investigators, and organizations. One of the preconditions for addressing this problem is to have a commonly used data representation to describe laboratory protocols that is unambiguous enough for precise interpretation and automation, yet simultaneously abstract enough to support reuse and adaptation.

Authors' addresses: Bryan Bartley, bryan.bartley@raytheon.com; Jacob Beal, jakebeal@ieee.org; Miles Rogers, miles.rogers@raytheon.com, Raytheon BBN Technologies, Cambridge, MA, USA; Daniel Bryce, dbryce@sift.net; Robert P. Goldman, rpgoldman@sift.net, SIFT, LLC, Minneapolis, MN, USA; Benjamin Keller, bjkeller@uw.edu, University of Washington, Seattle, WA, USA; Peter Lee, plee@ginkgobioworks.com, Ginkgo Bioworks, Cambridge, MA, USA; Vanessa Biggers, vanessa.biggers@strateos.com; Joshua Nowak, josh.nowak@strateos.com, Strateos, Inc., Menlo Park, CA, USA; Mark Weston, weston@netrias.com, Netrias, Inc., Cambridge, MA, USA.

While there has been much prior work on representations for protocols, prior approaches have generally been limited either by their dependence on natural language or in the expressiveness of their representation. Many protocol representations focus on simplifying the capture and distribution of descriptions in natural language, such as protocols.io [26] and the many commercial electronic laboratory notebook products. A similar approach is used for recording protocol execution information with minimum information standards such as MIAME [6], MIFlowCyt [13], and STRENDA [27]. Much of the key information in such approaches is encoded using natural language, which is easier to solicit from experimentalists, but cannot be readily interpreted by machines. As a consequence, protocols and protocol execution records captured with such representations cannot be automatically validated and are often ambiguous, incorrect, or lacking key information. For the same reasons, such protocols cannot generally be executed by laboratory automation systems.

//**Note from rpg:** In the above, it is not clear whether "minimum information standard" is a term of art, in which case it should be defined, or if it is just intended to indicate that these are stripped-down languages.//

Other protocol representations have focused specifically on automation-assisted execution. In many cases, these are highly specific solutions tied to specific hardware, often proprietary and tied to particular vendors. Some representations have been made applicable to a broader set of automation systems, however, such as Autoprotocol [16] and Antha [24], or instead use laboratory technicians as their automation, as in the case of Aquarium [11]. All of these representations, however, have been generally "low level" in their description of protocols, focusing on very specific details of each operation. This specificity, which enables automated execution, on the other hand poses barriers to adoption and to generalization and reuse, since this level of detail obscures understanding, and hampers transfer to different environments. Such representations are typically it difficult to translate into more "human-friendly" forms.

Finally, there are a number of workflow languages that solve similar problems in business logic or information processing, such as UML [18], the Common Workflow Language [2], Taverna [29], and Cromwell [7], not to mention biology-specific workflow systems such as Toil [28] and Galaxy [10]. Their execution models in some cases are general enough to be applied to the description and execution of laboratory protocols, but to the best of our knowledge such an application has not previously been implemented. Further, their very generality makes it more difficult for a domain expert to see how to apply them to protocols: there's too much of a gap between abstract task execution and the tasks that must be performed in a laboratory.

We provide a unified approach to protocol representation that addresses all of these disparate needs and bridges prior approaches through the development of the Protocol Activity Markup Language (PAML), a free and open protocol representation building on a foundation of UML [18], Autoprotocol [16], and SBOL3 RDF [3, 15]. In section 2, we elaborate on the design goals for PAML, its three foundations, and the approach taken for implementation. Section 3 outlines the key elements of PAML's representation for protocols, libraries of primitive actions, and execution records. Section 4 then discusses the current prototype implementation of PAML and the execution environments it supports, and Section 5 discusses ongoing plans for development.

## 2 ARCHITECTURE

We begin by elaborating a minimal set of design requirements that we argue will be necessary for any broadly applicable representation of biological protocols. These requirements then led us to identify a set of core representational ingredients that together provide a sufficient foundation for addressing these requirements.

### 2.1 Design Requirements

Information about laboratory protocols is used for a wide range of purposes in research and development, at many different stages of experiment design, execution, data analysis, interpretation, and communication and sharing with other groups. As such, to be an effective as a broadly shared community standard, we argue that any protocol representation will need to be able to support at least the following goals regarding biological experimentation:

- **Execution of a protocol by either humans or machines:** When available, laboratory automation can greatly improve the productivity of researchers, so protocols should be specified in sufficient detail to enable them to be mapped for machine execution. At the same time, many laboratories do not have automation available and even when automation is available it is common for protocols to incorporate both automated and manual stages, so protocols also need to be presented in a succinct and human-friendly form.
- **Maintaining execution records and associated metadata markup:** When a protocol is actually executed, it is important to be able to record the specific times and places of execution, equipment used, etc. A protocol representation thus needs to include means of creating persistent data structure that records an instance in which a protocol is executed. Likewise, it is also important to support automatic metadata tagging of data that are collected in the course of protocol execution, insofar as the execution environment of a particular lab is able to support this. For example, a protocol that collects flow cytometry data on samples of different strains under varying growth conditions, it would be valuable support automatic association of each FCS file produced by the flow cytometer with the strain, growth conditions, time of data collection, calibration, etc. of the sample from which the FCS file was produced.
- **Mapping protocols from one laboratory environment to another:** Protocol replication and reuse requires the ability to map a protocol from one laboratory to another, despite their differences in the specific equipment, inventory, and information systems. A protocol representation cannot guarantee that a protocol can be transferred, particularly one that is poorly understood or delicate in execution. Rather, a specification should allow a protocol to say, to the best of the authors' knowledge, how to predict if a mapping will product a correct execution and how to check if an execution should be considered correct, i.e., what a protocol specification truly requires, which aspects can safely be varied, which must be honored, and what reasonable tolerances are for inputs and outputs.
- **Recording modifications of protocols and the relationship between different versions:** Protocols are likely to be the subject of ongoing improvement and maintenance. For example, a protocol may be modified in order to enable the protocol to be simpler to execute or more reliable, be executed at a lab with different equipment from the lab at which it was originated, to enable the protocol to be scaled up or scaled down, etc. A use pattern that may end up being common is to have a protocol be "too strict" (too specific) to be instantiated in a lab, and thus need modification. Rather than the modification being an alternative protocol, however, it would be preferable to generalize the protocol to allow it to run both where it could before and also in the new lab. This improved version could then be contributed back to be released as an updated version of the existing protocol.
- **Verification and validation of protocol completeness and coherence:** Authoring a protocol requires substantial care and effort, and the usefulness of the protocol can be compromised if its specification is ill-formed, erroneous, or incomplete. Supporting protocol authors in achieving correctness is thus an important goal for

a protocol representation, and while the implementation will depend on specific tooling, the representation specification must provide guidance as to what it means for a protocol to be complete, consistent, etc.

- **Planning, scheduling, and allocation of laboratory resources:** Laboratory resources are valuable, and some organizations will want to be able to optimize their use. To do so, a protocol representation should support (at least) extraction of resource requirements from activities in the protocol, and estimated durations. Note that the specifics of resource requirements and duration estimates are required will likely be a function of both the protocol and the lab. Which resources are limited, and must be considered in a planner or scheduler, versus those that can be effectively treated as unlimited, will vary by lab. Management styles will also vary between organizations.

### 2.2 Foundations: UML, Autoprotocol, SBOL RDF

In developing the PAML protocol representation, we adopted a principle of building upon existing standards wherever possible, in order to increase compatibility and interoperability, take advantage of existing tooling, and make the implementation of the as lightweight as possible.

*2.2.1 UML Behavior Models.* To this end, we began by identifying an established standard for workflow modeling that could provide both a well-defined and general formal semantics, yet also be sufficiently abstract to allow succinct expression and adaptation. We found such a model in Unified Modeling Language (UML) behavior representations (specifically the current version 2.5.1 [18]). UML behaviors provide a highly general domain-independent workflow model. This model has a formal execution model associated with it based a token-passing, which can support serial, parallel, non-deterministic, and distributed execution. Furthermore, as UML usages are often focused on diagram-based communication, it also provides a set of flow control and abstraction constructs that are useful for succinct and human-friendly communication about complex workflows. At the same time, the formal execution model provides an unambiguous semantics readily tractable for verification, validation, and other forms of machine reasoning.

*2.2.2 Autoprotocol Laboratory Primitives.* Being domain-independent, however, UML does not provide any guidance on primitive behaviors suitable for expressing laboratory-independent behaviors. For this, we turn to Autoprotocol [16]. Autoprotocol describes biological protocols in terms of a sequence of instructions, and while this linear workflow is not expressive enough for our requirements, the instructions themselves are primitives that can be readily mapped from one laboratory environment to another. These instructions, such as `liquid_handle`, `incubate`, `provision`, and `spin` have been specifically designed and refined by the authors of Autoprotocol specifically for their value as a basis set for expressing the activities of common biological protocols in a manner readily transported between different pieces of laboratory automation. The Autoprotocol instructions thus provide an reasonable starting point for building a library of laboratory-independent primitive behaviors. Some activities expressed using these primitives are at a lower level than would be desirable for a human experimenter, such as specifying pipette mixing as a sequence of repetitive liquid handling operations. Such patterns, however, can be readily captured as higher-level behavior primitives suitable for human operators but expandable into lower-level instructions.

*2.2.3 SBOL 3 Materials, Records, and RDF.* While UML models processes, it does not actually provide a representation to actually capture executions and associate traces with data. For this, we turn to the Synthetic Biology Open Language (SBOL) version 3 [3], which uses Semantic Web practices and resources, such as *Uniform Resource Identifiers* (URIs) and

ontologies, to unambiguously identify and define biological system elements. and to provide serialization formats for encoding this information in electronic data files.

While the early versions of SBOL focused only on genetic designs, it has since been expanded to represent and link information throughout the design-build-test-learn workflow [15]. SBOL provides succinct representations for all of the materials that would be used by a typical biological protocol—strains, reagents, media, experimental sample designs, etc.—along with the ability to track and distinguish between specific physical aliquots and replicates. On the input side of a protocol, SBOL's combinatorial design specifications [22] offers the ability to compactly specify combinations of experimental conditions. SBOL also incorporates the Ontology of Units of Measure (OM) [21] for specifying and recording measurements, as well as the W3C Provenance Ontology (PROV-O) [17] for linking specifications, samples, and data via traces of activity records. This last is precisely complementary to our selection of UML behaviors for representing workflows, as PROV-O leaves the actual definition of activities to users. Thus, we can use PROV-O as the basis for capturing execution traces, with the activities in the trace defined using the UML data model, built from laboratory primitives based on Autoprotocol, and the inputs, outputs, and data relations encoded using SBOL 3.

SBOL's semantic web basis also means that it is readily extensible, unlike UML or Autoprotocol. For this reason, we select SBOL RDF as the underlying data model for PAML and convert the relevant portions of UML and Autoprotocol into SBOL RDF extensions. This approach also allows PAML to be extended with additional custom information for particular uses and deployments. Critically, SBOL RDF also provides a partial closure reasoning model that allows much stronger and more "object-oriented" reasoning than plain RDF or OWL, while still allowing documents to reference external material. This allows for an intuitive chunking and linking of information, for example, being able to store and reason about a complete record of a protocol execution that has links to the protocol without being required to store a copy of the protocol in the same document.

Finally, SBOL RDF also offers a natural approach to management of protocol modifications and versioning, since SBOL RDF can be serialized into the sorted N-triples RDF format. This format is a stable serialization that can be readily differenced and inspected with standard version control tooling. Thus, implementing PAML using SBOL RDF allows protocols to be maintained by distributed communities of contributors using standard software development version control such as git or mercurial, and the larger ecosystem of associated community and projects management tooling.

## 3 PAML DATA MODEL

Following the architecture presented in Section 2, the PAML data model is formulated as an extension of SBOL 3, implemented by encoding an ontology for the UML behavior model and its supporting classes, plus additional classes for linking this information into PROV-O records, libraries of primitive UML activities based on Autoprotocol, and additional classes for tracking laboratory samples and data. In this section, we present all of the major concepts and classes of PAML; additional supporting classes and the complete current specification can be found in the PAML draft specification at http://bioprotocols.org.

Following the pattern of SBOL, PAML's data model is defined in terms of classes, which can be instantiated as objects. Classes contain data in the form of properties, which may have primitive XML types (string, integer, long, float, boolean, or URI) or may refer connect to another object via its URI. Diagrams for the data model use UML conventions, in which classes are shown as rectangles labelled at the top with their class name, primitive properties are shown in text within the rectangle, object properties shown via an arrow with a diamond at the end, and class inheritance by arrows with empty heads. For object properties, filled diamonds indicate association properties, in which an object "owns" a child object, while empty diamond indicate references to an object defined elsewhere. Finally, the number of values a

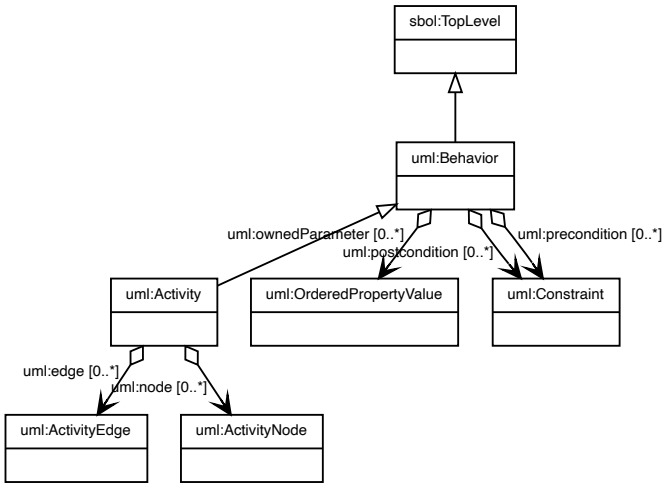



Fig. 1. UML `Activity` and `Behavior` classes. In PAML, a `Protocol` is defined as a UML `Activity`, while a `Primitive` laboratory action is defined as a UML `Behavior`.

property can have is indicated by a cardinality, with $[1]$ indicating a required property, $[0 \dots 1]$ indicating an optional property, $[0 \cdots *]$ indicating a property that can have any number of values, and $[1 \dots *]$ a property that must have at least one value.

To illustrate the data model in this section, we will use a running example of the iGEM LUDOX protocols for calibration of plate reader optical density (OD) measurements [23], introduced in the 2016 iGEM interlaboratory styd [5]. This is an extremely simple protocol consisting of three steps: water is added to four wells in a 96-well plate, LUDOX silica suspension is added to another four wells, and then all eight samples are measured at some specified absorbance (600nm in its original usage), in order to obtain a baseline measurement of OD, validate machine behavior, and allow path-length correction.

### 3.1 Protocols

In UML 2.5.1 [18]), the basic building block of process modeling is a `Behavior`, which is an abstract specification for how the state of a system changes over time. An `Activity` is a type of `Behavior` that defines a process in terms of a network of steps linked together by flows of information and control. For PAML, we import a strict subset of UML 2.5.1 into SBOL RDF, omitting those classes and properties that are not currently necessary for PAML.

*3.1.1 Laboratory Primitives are UML Behaviors.* Figure 1 shows the class structure for the adaptation of `Behavior` and `Activity` into SBOL RDF. A UML `Behavior` provides the definition an interface for a process. The `ownedParameter` property links to an ordered list of `Parameter` objects, which describe the order and type of the input arguments that can be given when the `Behavior` is invoked and of the output values which will be returned when the `Behavior`

completes its execution. Additional optional `precondition` and `postcondition` properties provide `Constraint` objects that specify requirements on `Parameter` values before and after execution, respectively.

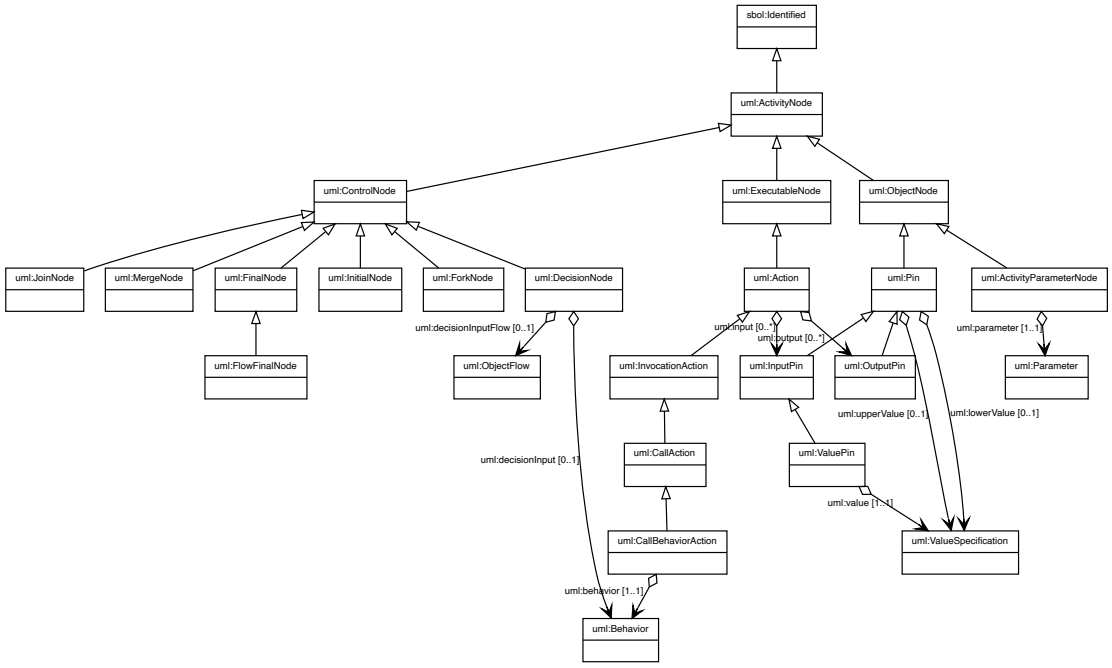
In PAML, the is simplest building block for a protocol is a `Primitive`, which are basic laboratory operations such as pipetting, measuring absorbance in a plate reader, or centrifuging, and which is defined simply as a UML `Behavior`. These object do not provide any information on how to actually carrying out actions like pipetting. Instead, they serve as the handoff point between PAML and an execution environment that knows how to actually carry out such primitives in a laboratory, as described further in Section 4.3.

*3.1.2 Sample Collections and Primitive Libraries.* In order to build useful protocols in PAML, we also need libraries of primitives that are simple enough to be readily reused, yet simultaneously abstract enough to be readily transferred from laboratory to laboratory. As noted above, Autoprotocols [16] already provides a good beginning set of such primitive operations, which have already been validated as both readily reused and transferrable between different pieces of laboratory equipment. In adapting Autoprotocol, however, PAML adds two key extensions to make protocols more adaptable and reusable: classes for describing collections of samples and organization of primitive operations into libraries.

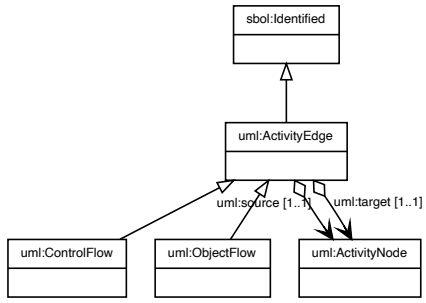
In Autoprotocol it is only possible to address one location at a time, i.e., a single container or a single compartment within a container, such as a well on a plate. This means that any operation on multiple locations must name every each location as a fixed value in the definition of the protocol, which in turn means that protocols are often both extremely large (e.g., individually operating on every well of a 96-well plate) and rigid, since locations cannot be supplied as a parameter value or determined dynamically at runtime.

In common practice, however, protocols are often naturally described in terms of operations on physical or logical collections of samples, such as "Wells A1 to D2 in a standard 96-well plate", "A 6x3 group 10ml tubes: three replicates each for six conditions," or "All wells showing green fluorescence >500 MEFL." Being able to represent such descriptions directly allows representations to be more compact, more intelligible to humans for both authoring and execution, and also more reusable, since they can communicated as parameters or determined dynamically. PAML thus includes a `SampleCollection` class, as shown in Figure 3(a), that allows it to represent such a collection directly, either as an N-dimensional `SampleArray` (e.g., the wells of a 96-well plate, a sequence of 10 flasks) or as a `SampleMask` that selects a subset of such an array with mask of Boolean values.

In particular, A `SampleArray` specifies an n-dimensional rectangular array of samples, all stored in the same type of container. For example, a `SampleCollection` might describe a set of 10 cell cultures growing in 96-well plate wells, or a set of 6 streaked agar plates, or a single 500 mL flask filled with media. For any non-empty location, the contents are in turn described by an SBOL 3 `Implementation` object that represents a physical sample and which, in turn, can link to an SBOL 3 `Component` object that describes the mixture of materials in that location. Note that this is a logical array, and does not necessarily indicate the actual layout of the samples in space, which may be determined by context in the execution environment. For example, a 2x4 array of samples in 96-well plate wells might end up being laid out as a 2x4 array in wells A1 to B4 or as a 2x4 array in wells G5 to H8 or as an 8x1 column in wells A1 to H1, or even as eight wells scattered arbitrarily around the plate according to an anti-bias quality control schema. This also allows for higher-dimensional arrays where each dimension represents an experimental factor. For example, an experiment testing four factors with 3, 3, 4, and 5 values per factor, for a total of 180 combinations, could be represented as a 4-dimensional `sample` array of 96-well plate wells, and then end up laid out over two plates.

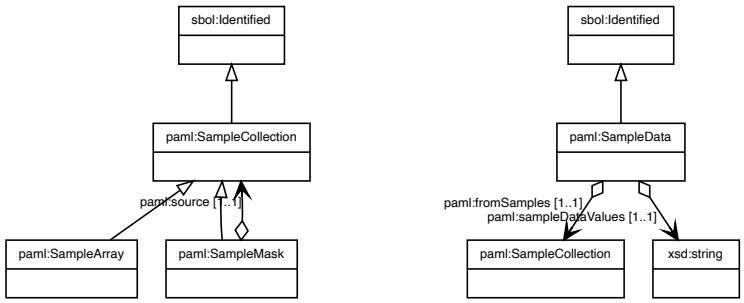



(a) UML ActivityNode




(b) UML ActivityEdge

Fig. 2. A UML `Activity` is defined as a network of `ActivityNode` objects that define steps, decisions, or values in the activity and `ActivityEdge` objects that connect between them.




(a) PAML SampleCollection (b) PAML SampleData

Fig. 3. PAML represents collections of samples as either N-dimensional arrays or using Boolean masks to select subsets from such an array (a) and data in terms of an association of an array of values with a sample collection (b).

Complementarily, a `SampleMask` describes a subset of samples in some `SampleCollection` (array or mask) by an array of Boolean values, where true values indicate that a sample is included and false values indicate that it is excluded. To allow masks to be readily composed and interchanged, their dimension is kept identical to that of the source `SampleCollection`. In this way, references to the physical laboratory elements on which a `SampleArray` is laid out can be easily maintained and combined even across different subsetting operations.

Finally, PAML defines one more class, `SampleData`, as shown in Figure 3(b), in order to capture the relationship between physical samples and information about those samples. The `SampleData` class simply associates a `SampleCollection` with an array that has values defined for all of the included samples of the collection, thereby representing measurements, such as an array of plate reader absorbance measurements. With the addition of the `SampleCollection` and `SampleData` classes, PAML primitives can thus be defined in terms of operations on collections of samples, rather than on specific locations, e.g., dispensing media into a collection of wells or measuring the absorbance in those wells.

PAML also extends the concepts in Autoprotocol with the addition of a notion of libraries for organizing collections of primitives. In Autoprotocol there is a fixed set of primitives defined by the specification, which limits extensibility while at the same time constraining execution environments to those that can support the full set of primitives. For PAML, we take an approach more like that used in most modern programming languages, in which the specified language is kept as small as possible, while much of the core capabilities of the language are provided by collections of functions grouped into libraries, with each library associated with a well-defined cluster of functionality.

Figure 4 shows how the current implementation of PAML organizes operations from Autoprotocol into four libraries of PAML `Primitive` objects. One library is mostly new operations for creating and subsetting `SampleCollection` objects. The other three libraries are classes laboratories activities that operate on `SampleCollection` objects. The `plate_handling` library contains operations performed on containers (e.g., plates, flasks), such as sealing and incubation, which are mostly direct mapping of equivalent Autoprotocol activities. The `liquid_handling` library contains operations moving liquids within or between containers, such as pipetting from one location to another, or using a pipette to

| PAML Library/Primitive | Autoprotocol Equivalent |
|---|---|
| **Library: `sample_arrays`** | |
| `EmptyContainer` | Undocumented "SUPPLY NEW CONTAINER" operation |
| `PlateCoordinates` | *n/a - Autoprotocol references only single locations* |
| `Rows` | *n/a - Autoprotocol references only single locations* |
| `Columns` | *n/a - Autoprotocol references only single locations* |
| `DuplicateCollection` | *n/a: operation makes an array with the same shape as an input* |
| `ReplicateCollection` | *n/a: DuplicateCollection, adding a new replicate dimension* |
| **Library: `plate_handling`** | |
| `Cover` | `cover` |
| `Incubate` | `incubate` |
| `Seal` | `seal`, flexible mode |
| `AdhesiveSeal` | `seal`, `mode=thermal` |
| `ThermalSeal` | `seal`, `mode=adhesive` |
| `Spin` | `spin` |
| `Uncover` | `uncover` |
| `Unseal` | `unseal` |
| **Library: `liquid_handling`** | |
| `Provision` | `provision` |
| `Dispense` | `liquid_handle`, `mode=dispense` |
| `Transfer` | `liquid_handle` up in one location, down elsewhere |
| `TransferInto` | `liquid_handle` with a non-empty destination |
| `PipetteMix` | `liquid_handle` up and down repeatedly in same location |
| **Library: `spectrophotometry`** | |
| `MeasureAbsorbance` | `spectrophotometry`, `mode=absorbance` |
| `MeasureFluorescence` | `spectrophotometry`, `mode=fluorescence` |
| *Currently unimplemented* | `acoustic_transfer`, `flow_cytometry`, `measure_mass`, `measure_volume`, `spectrophotometry mode=luminescence/shake` |

Fig. 4. PAML's `Primitive` laboratory operations are organized into libraries based on required equipment types. The initial collection of primitives are based on Autoprotocol, currently implementing most functionality from that language, plus primitives that generalize references from individual containers to collections of samples.

mix fluids in a location. This includes a division of the omnibus Autoprotocol `liquid_handle` operation into several different patterns of usage, in order to make higher-level abstractions that are both more readily human-interpretable and also more readily accessible for machine reasoning and verification. The final library, `spectrophotometry`, does the same for plate reader measurements.

Organizing primitives into libraries and aligning libraries with equipment also provides a basis for comparing protocol requirements and laboratory capabiltiies. A protocol's capability requirements may be coarsely determined by the set of libraries that it uses. Any given protocol execution environment can then be defined in terms of which libraries are supported. Moreover, different libraries may be supported with different means of execution. For example, a laboratory with a liquid handling robot and a plate reader may support automated execution of `Primitive` operations from the `liquid_handling` library, while those in `plate_handling` and `spectrophotometry` are carried out by a human operator.

*3.1.3 Protocols are UML Activities.* We now move from individual laboratory operations to complete protocols. In UML, an `Activity` is used to define composite behaviors in terms of a network of steps. Its `node` property stores a

collection of ActivityNode objects (Figure 7(a) in three types: CallBehaviorAction, ObjectNode, and ControlNode. A CallBehaviorAction object defines an actual steps of the protocol, in which a specific Primitive or sub-Protocol is carried out (note that this class is several layers deep in a hierarchy of other sibling UML ExecutableNode classes not currently used in PAML). Complementarily, an ObjectNode defines a point where data enters and exits the Activity (ActivityParameterNode) or one of its CallBehaviorAction nodes (Pin, with constants using ValuePin). Finally, a ControlNode is used to define where the steps of the Acitivity start (InitialNode, stop (FinalNode), branch (splitting at a DecisionNode and rejoining at a MergeNode), or run in parallel (splitting at a ForkNode and rejoining at a JoinNode). The ActivityEdge objects that connect pairs of ActivityNode objects, by contrast, are much simpler, merely indicating a path by which either a control or object token flows from a source to a target.

The execution semantics defined for an Activity are based on a notion of token flow similar to Petri nets [20]. When the Activity begins execution, tokens start in any InitialNode or input ActivityParameterNode. From there, they flow along the connected ActivityEdge for each such source node to the connected target node. As tokens arrive at an ActivityNode, it waits until a token has arrived along every edge for which it is a target; once all edges have delivered a token, the ActivityNode takes action if needed (i.e., it is a CallBehaviorAction), then sends a token along every edge for which it is a source. The exceptions are DecisionNode, whose purpose is to choose one of several edges on which to send a token, and JoinNode, A CallBehaviorAction node, on the other hand, also waits for incoming tokens to the InputPin objects that define values for the input parameters of the Behavior it will call, and also sends tokens from its associated OutputPin objects. This process continues, with tokens proceeding along nodes and edges until execution comes to an end as tokens are absorbed by FinalNode and/or output ActivityParameterNode objects.

All told, this implements a universally expressive model of behavior execution. The control flow semantics can support ordered steps, via ObjectFlow edges linking pins and ControlFlow edges linking steps, as well as steps in parallel or in arbitrary order, via ForkNode and a lack of constraining edges. Execution patterns can include loops, by means of DecisionNode and circular edge patterns, and recursions, by an Activity with a CallBehaviorAction that calls itself. Furthermore, since tokens can potentially be communicated between different agents and different types of agent, this model also allows for execution to be distributed, e.g., between several pieces of automation equipment, as a mixture of human and automated execution, or even across a group of collaborating laboratories.

UML Activity is used to implement the core concept of PAML, the Protocol. A Protocol could be very simple, such as the iGEM LUDOX calibration protocol, as shown in Figure 5. In this case the Protcol consists of 11 ActivityNodes and 15 ActivityEdges connecting them together into a network. Together, they implement a Behavior with an interface of one input Parameter—the wavelength to be measured—and one output Parameter, the absorbance measured at the plates.

In this implementation, two nodes initiate the execution of the protocol: the InitialNode and the ActivityParameterNode for the wavelength input parameter. To manage the ordering of the steps, ControlFlow edges are added. The first of these links from the InitialNode to a CallBehaviorAction to invoke EmptyContainer from the sample_arrays library, requesting allocation of a SampleArray for a 96-well clear flat-bottom plate (specified by the plateRequirement input not expanded in the visualization). This in turn enables three CallBehaviorAction nodes for PlateCoordinates operations from the same package, which each create a SampleMask selecting a set of wells within the plate, with a ForkNode implicitly inserted to support the same plate being accessed multiple times. The next nodes are the three CallBehaviorAction nodes that actually do "real" lab work: two instances of the Provision primitive from the liquid_handling library put water and LUDOX into the plate, respectively, and an instance of the MeasureAbsorbance primitive from the spectrophotometry library measures their absorbance at the specified wavelength. Note that the

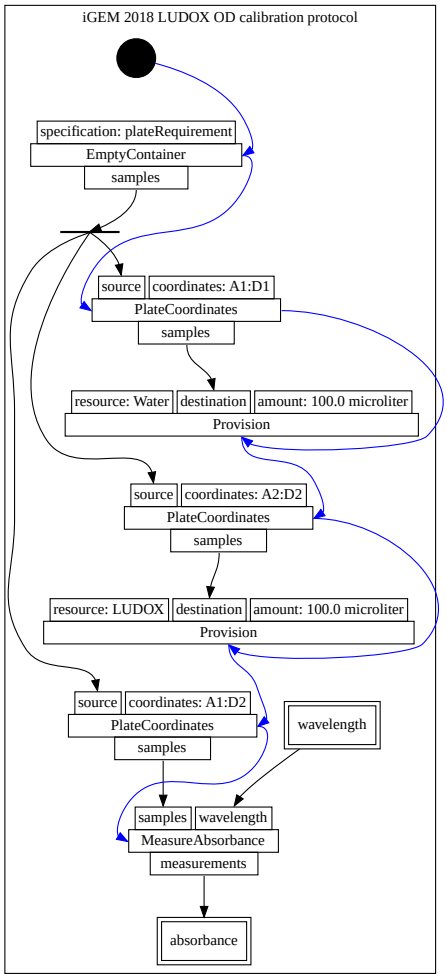



Fig. 5. PAML prototocol for the iGEM 2018 LUDOX calibration protocol. The graph includes protocol activities that follow a control flow denoted by blue edges, and data flow denoted by black edges. The graph also illustrates protocol input (e.g., wavelength) and output (e.g., abosorbance) parameters with double boxes.

`Provision` operations do not produce outputs, instead acting by modifying state of the `SampleCollection` provided by their `destination` input. As such, it is necessary to add control edges that ensure that the water is added before the LUDOX, and that both have been put in the plate before absorbance is measured. Finally, once the lab work has been carried out, the final output `ActivityParameterNode` collects and reports the `SampleData` that is output from the call to `MeasureAbsorbance`.

Since a `Protocol` is itself also a `Behavior`, it can also be used to implement a much more complex protocols, including ones with complex hierarchical structures. For example, a `Protocol` might specify a cell culturing protocol that invokes media adjustment and data collection sub-protocols at various time points, or a multi-stage DNA assembly protocol involving multiple rounds of digestion, ligation, transformation, and selection. Furthermore, requesting the execution

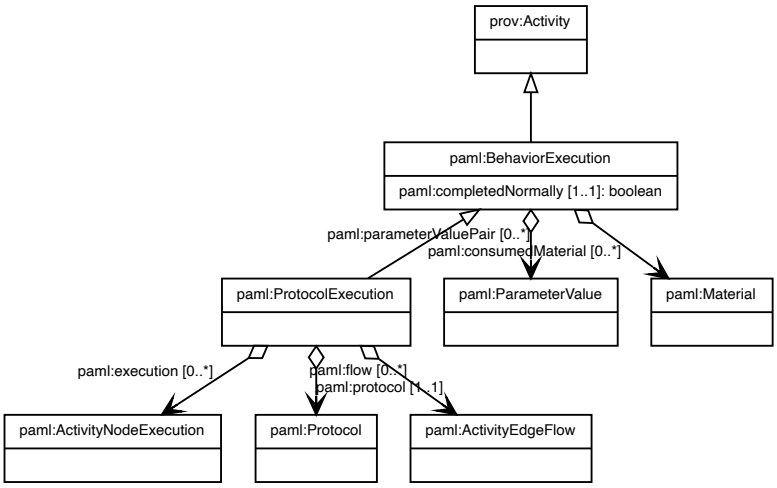



Fig. 6. PAML `ProtocolExecution` and `BehaviorExecution` classes, used for recording the execution of PAML `Protocols` and `Primitives`.

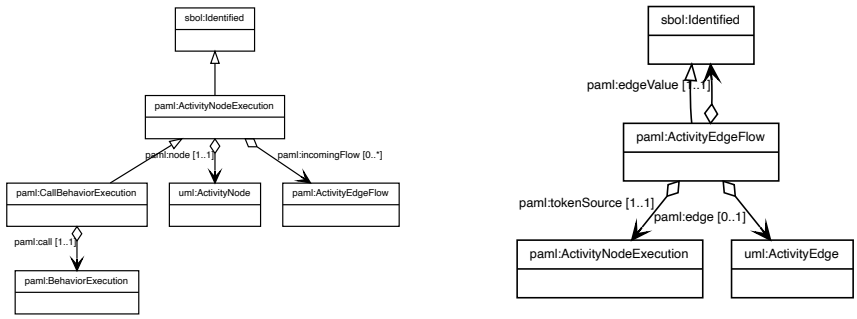
of a `Protocol` on a particular set of samples or conditions can itself be represented as a simple "wrapper" `Protocol` in which the desired sample and condition values are supplied as `ValuePin` inputs on a `CallBehaviorAction` invoking the `Protocol` and its results are collected to output via an `ActivityParameterNode` for each expected result. In principle, even an entire experimental campaign could be encoded in such a manner, if desired.

### 3.2 Execution Records

PAML's representation for recording the trace of an execution is built on the W3C provenance ontology (PROV-O) [17], which provides a mechanism for recording traces. PROV-O has its own distinct notion of an `Activity`, in this case a record of an instance of an execution. The basic notion is thus that an execution trace consists of a structure of PROV-O `Activity` objects, each linked to a corresponding UML `Behavior` object specifying the type of the activity.

The PROV-O `Activity` is only a stub class, however, so to provide additional information needed for recording information about protocol executions, PAML extends PROV-O `Activity` with a `BehaviorExecution` class for recording execution of a UML `Behavior` and a child `ProtocolExecution` class for recording execution of a UML `Activity`, as shown in Figure 6.

A `BehaviorExecution` is a record of how a `Protocol`, `Primitive`, or other `Behavior` was carried out. Such an execution might be either real or simulated, for example as part of "unrolling" a protocol for fixed execution environment as described below in Section 4.3. Properties inherited from PROV-O the basic structure of the trace: the PROV-O `type` property links to the `Behavior`, its `startedAtTime` and `endedAtTime` properties record timing information, and the entity carrying out the execution (e.g., a particular person in the lab, a liquid handling robot, a plate reader) is recorded with a PROV-O `Association` to a PROV-O `Agent`. Additional PAML-specific properties records the input and output




(a) UML ActivityNode (b) UML ActivityEdge

Fig. 7. A UML `Activity` is defined as a network of `ActivityNode` objects that define steps, decisions, or values in the activity and `ActivityEdge` objects that connect between them.

`Parameter` values for the `Behavior`, the laboratory materials consumed (as pairs of an SBOL `Component` specifying material type with a `Measure`), and whether the execution completed normally or if there was some exception condition.

A `ProtocolExecution` extends this with the addition of records for the nodes and edges defining the `Protocol`'s behavior as a UML `Activity`. Specifically, the execution property is used to record each firing of a `ActivityNode` and the flow property is used to record each time a token moves along a `ActivityEdge`, using the `ActivityNodeExecution` and `ActivityEdgeFlow` classes shown in Figure 7. In the case of invocation of a `Primitive` or `Protocol` via a `CallBehaviorAction`, a `CallBehaviorExecution` provides a link down to the `BehaviorExecution` sub-trace that records the execution of the `Primitive` or `Protocol`.

For a simple protocol without any branches or sub-protocols, such as the iGEM LUDOX protocol described above, there will be precisely one execution for each node or edge. For more complex protocols there may be none on branches not taken or multiple in the case of looping constructs. In either case, however, this parallel construction provides the necessary representation for recording information about the execution of protocols in the lab.

## 4 PROTOTYPE

### 4.1 Ontology-Based Specification

The PAML data standard is specified first and foremost as an ontology encoded in the Web Ontology Language (OWL). We leverage this machine-readable specification to make the standards development process more efficient through automation. First, we automatically generate graphical visualizations of the data model which illustrate the classes, properties, and links between classes. Second, we automatically generate a specification document as LaTeX. This human-readable document incorporates the class diagrams as well as explanatory descriptions which are sourced from annotations contained in the original ontology file. Finally, we automatically generate an object-oriented Python API which supports authoring and exchange of PAML protocols as any of a number of standard RDF formats. We use a tool called SBOLFactory [4] to dynamically generate this Python class framework directly from the ontology file. This approach can be generalized to support development of other custom extensions to the SBOL data model besides PAML.

Moreover, this approach could be adapted to support standards development efforts in other knowledge domains as well.

We use ontology-based specification because it has several advantages. As the standard is still in an early stage of development and is expected to evolve, revisions to the proposed data model can be rapidly generated, released, and actually tested in practice. Moreover, since the human-readable specification document and the software library are generated from a single source, we avoid introducing errors and discrepancies between the different artifacts. Overall, these factors enable rapid development and responsiveness of PAML developers to emerging needs and use cases from the open protocols community.

The Python API also performs validation on protocols to ensure that representations are complete and consistent. For example, one such validation rule requires that every ProtocolExecution must link to a Protocol. We use the Shapes Constraint Language (SHACL) [12] to encode these validation rules in a declarative syntax. SHACL is an RDF-based language that describes graph patterns to which valid data instances must conform. We use the pySHACL [25] validator tool to check conformance of all PAML instance data according to the encoded rules. By using the combination of OWL and SHACL, we can fully specify the PAML data model using non-ambiguous, machine-readable languages. The specification is thus decoupled from its implementation in any one programming language as well as its formulation in natural language.

### 4.2 Visualization and Editing of Protocols

Visualizing and editing PAML protocols is most natural as a graph because it is based upon the UML Activity model. Each protocol involves a set of activities and controls (nodes) that are linked by data and control flows (edges). Figure 5 illustrates the rendering of the iGEM 2018 LUDOX calibration protocol via GraphViz [9]. It includes an initial control node (filled black circle) that is followed by (denoted by a blue edge) an activity node EmptyContainer. The activity nodes include input pins (e.g., specification) and output pins (e.g., samples). Data flow edges (denoted by black edges) link the activity pins (e.g., EmptyContainer samples link to the PlateCoordinates source) either directly or through control nodes such as a fork node (denoted by a black bar). Data flow edges also link protocol input (e.g., wavelength) and output (e.g., absorbance) parameters.

While Figure 5 illustrates the protocol as a graph, PAML can be illustrated in a number of formats. The protocol illustrated by Figure 5 may also be described as a list of activities because the activities are totally ordered. PAML protocols that are partially ordered or include decision nodes can be represented by other paradigms such as block-based programs [14] or visual scripts [1, 8]. More broadly, protocols can be viewed as programs that can be expressed in a programming language or pseudocode.

//**Note from rpg:** In the following, the label “fig:paml_api” does not seem to be defined.//

In addition to visualizing protocols, the same paradigms support editing, such as adding, deleting, or configuring activities. The PAML Python API supports protocol editing operations by providing functions that build a protocol. Protocol generation scripts (e.g., as listed in Figure **??**) execute a sequence of API functions that construct the PAML protocol. Lines 5 to 7 define the protocol and add it to an SBOL3 document (representing the protocol as RDF). Line 10 defines an input parameter called wavelength. Line 13 defines the SBOL3 object for water, and links it to a PubChem identifier. Line 18 defines a microplate object that will hold the samples. Lines 21 to 24 identify the wells that will hold water and provision the water into those wells. Lines 29 to 32 identify which wells to measure and then measure the absorbance. Finally, lines 35 to 38 define the protocol output parameter for absorbance and links it to the output of the

```python
import sbol3
import paml

# declare a protocol and add it to an SBOL3 document doc (doc initialization omitted).
protocol = paml.Protocol('iGEM_LUDOX_OD_calibration_2018')
protocol.name = "iGEM 2018 LUDOX OD calibration protocol"
doc.add(protocol)

# add an optional parameter for specifying the wavelength
wavelength_param = protocol.input_value('wavelength', sbol3.OM_MEASURE, optional=True, default_value=sbol3.Measure(600,
      tyto.OM.nanometer))

# create the materials to be provisioned (Ludox omitted for brevity)
ddh2o = sbol3.Component('ddH2O', 'https://identifiers.org/pubchem.substance:24901740')
ddh2o.name = 'Water'
doc.add(ddh2o)

# get a plate (spec omitted for brevity)
plate = protocol.primitive_step('EmptyContainer', specification=spec)

# identify wells to use
c_ddh2o = protocol.primitive_step('PlateCoordinates', source=plate.output_pin('samples'), coordinates='A1:D1')

# put water in selected wells
provision_ddh2o = protocol.primitive_step('Provision', resource=ddh2o, destination=c_ddh2o.output_pin('samples'), amount=
      sbol3.Measure(100, tyto.OM.microliter))

# similar Ludox PlateCoordinates and Provision steps omitted

# identify wells to use
c_measure = protocol.primitive_step('PlateCoordinates', source=plate.output_pin('samples'), coordinates='A1:D2')

# measure the absorbance
measure = protocol.primitive_step('MeasureAbsorbance', samples=c_measure.output_pin('samples'))

# link input parameter to measure primitive input
protocol.use_value(wavelength_param, measure.input_pin('wavelength'))

# link measurement output to protocol output
output = protocol.designate_output('absorbance', sbol3.OM_MEASURE, measure.output_pin('measurements'))
```

Fig. 8. PAML Python script to construct a portion of the iGEM Ludox calibration protocol. An interactive Jupyter notebook that includes a full script is available at: https://colab.research.google.com/drive/1WPvQ0REjHMEsginxXMj1ewqfFHZqSyM8?usp=sharing

MeasureAbsorbance activity. Visual editors can use these functions to implement the same functionality as the Python script.

Visualizing protocols, like visualizing source code, helps to specify and understand the protocol. However, like programs, executing a protocol requires interpreting the steps. As the number of objects and control flow nodes increases, it becomes increasingly difficult to interpret the protocol. Furthermore, PAML protocol serialization to different target languages (e.g., Autoprotocol) may require compiling away some of the control structure. To support execution either directly or though serialization, we also developed an execution engine for PAML.

### 4.3 Execution in Markdown and Autoprotocol

PAML execution requires interpreting a protocol to determine which activity can be executed next, and then recording the data generated on the output pins. The PAML execution engine uses a token based execution semantics that implements the UML activity model based upon Petri-nets [20]. The execution involves tracking a set of tokens generated by each activity or control node in the protocol. Each activity generates tokens on their output pins, and

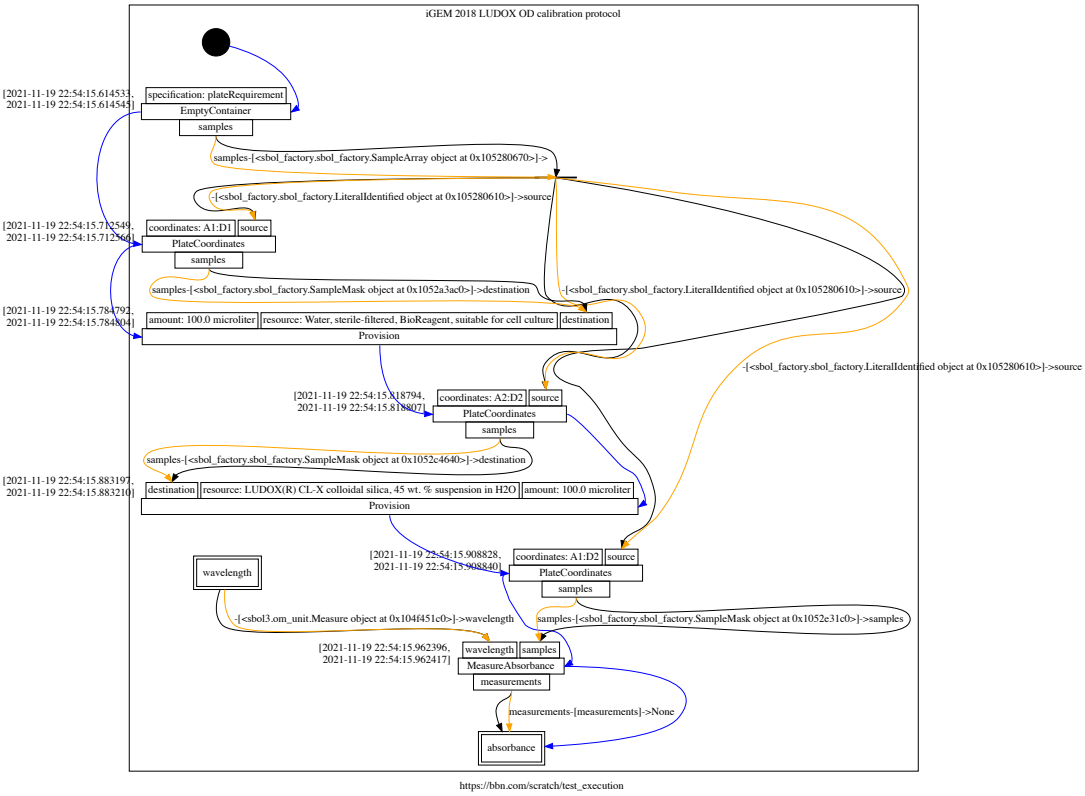



Fig. 9. PAML execution trace for the iGEM 2018 LUDOX calibration protocol, layered on the protocol visualization shown in Figure 5. Yellow edges denote data flow with placeholder and computed values for an offline execution of the protocol.

consumes tokens on the input pins. The tokens can either hold data representing objects created by activities, or denote control. The execution engine non-deterministically selects an enabled protocol node each iteration. It records the execution for each node in an execution trace and notifies any listeners. Executing a protocol offline involves generates identifiers for activity outputs that will be specified later, during an actual execution of the protocol. Online execution of the protocol provides an opportunity to specify actual outputs as the activities execute. Figure 9 shows an example visualizing an execution trace for the LUDOX protocol.

In order to generate serializations of protocols to alternative target formats, the execution engine uses listeners that output either Autoprotocol or markdown. Both Autoprotocol (a machine language) and markdown (a list of steps in semi-structured natural language) require a sequence of steps that totally order the protocol activities and omit control flow that is otherwise implicit in the format (e.g., omitting the object fork node for the plate, as illustrated by the black bar in Figure 5). The execution listener pattern is particularly helpful for serializing protocols that include multiple repetitions of sub-protocols as activities. This requires the protocol executor to “unroll” the protocol into a series of distinct executions activities appearing in a repeated sub-protocol.

# 1 IGEM 2018 LUDOX OD CALIBRATION PROTOCOL

## 1.1 Description:

With this protocol you will use LUDOX CL-X (a 45% colloidal silica suspension) as a single point reference to obtain a conversion factor to transform absorbance (OD600) data from your plate reader into a comparable OD600 measurement as would be obtained in a spectrophotometer. This conversion is necessary because plate reader measurements of absorbance are volume dependent; the depth of the fluid in the well defines the path length of the light passing through the sample, which can vary slightly from well to well. In a standard spectrophotometer, the path length is fixed and is defined by the width of the cuvette, which is constant. Therefore this conversion calculation can transform OD600 measurements from a plate reader (i.e. absorbance at 600 nm, the basic output of most instruments) into comparable OD600 measurements. The LUDOX solution is only weakly scattering and so will give a low absorbance value.

## 1.2 Protocol Materials:

- Water, sterile-filtered, BioReagent, suitable for cell culture[1]
- LUDOX(R) CL-X colloidal silica, 45 wt. % suspension in H2O[2]

## 1.3 Protocol Inputs:

- `wavelength` = 600.0

## 1.4 Protocol Outputs:

- `absorbance`

## 1.5 Steps

1. Provision a container named `samples` meeting specification: cont:ClearPlate and cont:SLAS-4-2004 and (cont:wellVolume some ((om:hasUnit value om:microlitre) and (om:hasNumericalValue only xsd:decimal[>= "200"^^xsd:decimal]))).
2. Pipette 100.0 microliter of Water, sterile-filtered, BioReagent, suitable for cell culture[3] into `samples(A1:D1)`.
3. Pipette 100.0 microliter of LUDOX(R) CL-X colloidal silica, 45 wt. % suspension in H2O[4] into `samples(A2:D2)`.
4. Make absorbance measurements (named `measurements`) of `samples(A1:D2)` at 600.0 nanometer.
5. Report values for `absorbance` from `measurements`.

Fig. 10. PAML protocol for iGEM 2018 Ludox calibration converted to markdown.

PAML is converted to Autoprotocol and markdown with different execution listeners. Figure 10 illustrates the converted and rendered markdown and Figure illustrates the converted Autoprotocol describing the iGEM Ludox calibration protocol. The listeners not only collect the translated sequence of protocol activities, but help to resolve the objects appearing the protocol. For example, the Autoprotocol listener interprets "EmtpyContainer" activity to identify a container (c.f., line 38 in Figure 11) in the Strateos LIMS that will satisfy the specification made in the protocol. Similarly, the markdown listener interprets the protocol to construct a human-readable strings that describe each step. In addition to interpreting the steps, the listeners format the syntax needed for each target language. The Autoprotocol listener formats protocols as a list of instructions in JSON, and the markdown listener makes use of markdown syntax to hyperlink definitions of reagents and containers.

# 5 FUTURE DIRECTIONS

The effort on PAML described above have produced a draft representation that appears to be simultaneously expressive enough and compact enough to meet the satisfy all of the key goals that we have identified for a broadly applicable community standard. Our prototype implementation realizes this representation in the form of an ontology, specification,

```
{"instructions": [
   { "op": "provision",
     "resource_id": "rs1c7pg8qs22dt",
     "measurement_mode": "volume",
     "to": [
       {
         "well": "samples/0",
         "volume": "100:microliter"
       },
       <Others omitted>
     ]},
   { "op": "provision",
     "resource_id": "rs1b6z2vgatkq7",
     "measurement_mode": "volume",
     "to": [
       {
         "well": "samples/1",
         "volume": "100:microliter"
       },
       <Others omitted>
     ]},
   { "op": "spectrophotometry",
     "dataref": "measurements",
     "object": "samples",
     "groups": [
       {
         "mode": "absorbance",
         "mode_params": {
           "wells": [
             "samples/0",
             <Others omitted>
           ],
           "wavelength": [
             "600:nanometer"
           ]}}]}],
 "refs": {
   "samples": {
     "id": "ct1g9qsg4wx6gcj",
     "discard": true
   }}}
```

Fig. 11. Autoprotocol specification of the iGEM Ludox calibration protocol generated by the PAML execution engine and Autoprotocol listener.

and Python library, which in turn have been used to implement test protocols and tools for visual editing and for execution, either by hand via export to a "paper protocol" or with laboratory robotics via export to Autoprotocol.

The next critical stage in developing this into an effective community standard for protocols is to refine the representation and expand the set of tools through involvement of interested stakeholders from the broader community. To that end, we organized an open community meeting at the COMBINE 2021 standards meeting in October, 2021, during the course of which participants validated community interest in this initiative, prioritized next steps for PAML, and began organization of an open pre-competitive community for its continued development, which may be found at http://bioprotocols.org/

The key near-term goals for the development of PAML, as currently priorities by this community, are thus:

- Putting PAML to use in ongoing interlaboratory collaborations within the stakeholder community.
- Implementation of additional key execution environments, such as the OpenTrons API [19] and protocols.io [26].
- Determining how the representation will work with sample arrays and sample data
- Implementing reasoning about the contents of samples.

- Improved user interfaces for protocol design, editing, and inspection.

If this nascent community is able to achieve these goals, particularly in using PAML to reduce the protocol-related challenges faced by existing interlaboratory collaborations, then it will form the basis for further development and utilization and, ultimately, may be able to establish an effective open standard representation for biological protocols, accelerating research and development across a broad range of fields and applications.

## ACKNOWLEDGMENTS

This work was supported by Air Force Research Laboratory (AFRL) and DARPA contracts FA8750-17-C-0184, FA8750-17-C-0231, and HR001117C0095. This document does not contain technology or technical data controlled under either U.S. International Traffic in Arms Regulation or U.S. Export Administration Regulations. Views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

## REFERENCES

[1] [n. d.]. Unity Visual Scripting. https://unity.com/products/unity-visual-scripting. (accessed 2020-11-18).

[2] Peter Amstutz, Michael R. Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, and et al. 2016. Common Workflow Language, v1.0. https://doi.org/10.6084/m9.figshare.3115156.v2

[3] Hasan Baig, Pedro Fontanarrosa, Vishwesh Kulkarni, James Alastair McLaughlin, Prashant Vaidyanathan, Bryan Bartley, Jacob Beal, Matthew Crowther, Thomas E. Gorochowski, Raik Grunberg, Goksel Misirli, James Scott-Brown, Ernst Oberortner, Anil Wipat, and Chris J. Myers. 2020. Synthetic biology open language (SBOL) version 3.0.0. *Journal of Integrative Bioinformatics* 17, 2-3 (2020). https://doi.org/doi:10.1515/jib-2020-0017

[4] Bryan Bartley. 2021. *pySHACL*. https://github.com/SynBioDex/sbolfactory

[5] Jacob Beal, Traci Haddock-Angelli, Geoff Baldwin, Markus Gershater, Ari Dwijayanti, Marko Storch, Kim De Mora, Meagan Lizarazo, Randy Rettberg, and with the iGEM Interlab Study Contributors. 2018. Quantification of bacterial fluorescence using independent calibrants. *PloS one* 13, 6 (2018), e0199432.

[6] Alvis Brazma, Pascal Hingamp, John Quackenbush, Gavin Sherlock, Paul Spellman, Chris Stoeckert, John Aach, Wilhelm Ansorge, Catherine A Ball, Helen C Causton, et al. 2001. Minimum information about a microarray experiment (MIAME)?toward standards for microarray data. *Nature genetics* 29, 4 (2001), 365–371.

[7] Broad Institute. 2019. *The Workflow Description Language and Cromwell.* https://software.broadin-stitute.org/wdl

[8] Zdena Dobesova. 2011. Visual programming language in geographic information systems. In *Proceedings of the 2nd international conference on Applied informatics and computing theory*. World Scientific and Engineering Academy and Society (WSEAS), 276–280.

[9] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. 2003. Graphviz and dynagraph: static and dynamic graph drawing tools. In *GRAPH DRAWING SOFTWARE*. Springer-Verlag, 127–148.

[10] Jeremy Goecks, Anton Nekrutenko, and James Taylor. 2010. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology* 11, 8 (2010), R86.

[11] Ben Keller, Justin Vrana, Abraham Miller, Garrett Newman, and Eric Klavins. 2019. Aquarium: The laboratory operating system version 2.6.0. (2019). https://doi.org/10.5281/zenodo.2583232

[12] Holger Knublauch and Dimitris Kontokostas. 2017. Shapes Constraint Language (SHACL). https://www.w3.org/TR/shacl/.

[13] Jamie A Lee, Josef Spidlen, Keith Boyce, Jennifer Cai, Nicholas Crosbie, Mark Dalphin, Jeff Furlong, Maura Gasparetto, Michael Goldberg, Elizabeth M Goralczyk, et al. 2008. MIFlowCyt: the minimum information about a Flow Cytometry Experiment. *Cytometry Part A: the journal of the International Society for Analytical Cytology* 73, 10 (2008), 926–930.

[14] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15.

[15] James Alastair McLaughlin, Jacob Beal, Göksel Mısırlı, Raik Grünberg, Bryan A Bartley, James Scott-Brown, Prashant Vaidyanathan, Pedro Fontanarrosa, Ernst Oberortner, Anil Wipat, et al. 2020. The Synthetic Biology Open Language (SBOL) version 3: simplified data exchange for bioengineering. *Frontiers in Bioengineering and Biotechnology* 8 (2020), 1009.

[16] Ben Miles and Peter L Lee. 2018. Achieving reproducibility and closed-loop automation in biological experimentation with an IoT-enabled Lab of the future. *SLAS Technology* 23, 5 (2018), 432–439.

[17] Paolo Missier, Khalid Belhajjame, and James Cheney. 2013. The W3C PROV family of specifications for modelling provenance metadata. In *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 773–776.

[18] Object Management Group. 2017. OMG Unified Modeling Language (OMG UML) Version 2.5.1. https/www.omg.org/spec/UML/.

[19] Opentrons. 2020. OT-2 Python Protocol API Version 2. https://docs.opentrons.com/v2/
[20] Carl Adam Petri. 1966. *Communication with automata.* Ph. D. Dissertation. Universitat Hamburg.
[21] Hajo Rijgersberg, Don Willems, Xin-Ying Ren, Mari Wigham, and Jan Top. 2021. Ontology of units of Measure (OM), version 2.0.31. http://www.ontology-of-units-of-measure.org/resource/om-2.
[22] Nicholas Roehner, Bryan Bartley, Jacob Beal, James McLaughlin, Matthew Pocock, Michael Zhang, Zach Zundel, and Chris J Myers. 2019. Specifying combinatorial designs with the synthetic biology open language (sbol). *ACS synthetic biology* 8, 7 (2019), 1519–1523.
[23] Paul Rutten, Richard Tennant, Jacob Beal, Traci Haddock-Angelli, Natalie Farny, Geoffrey Baldwin, Marko Storch, and Ari Dwijayanti. 2019. Calibration Protocol - OD600 Inter-equipment Conversion with LUDOX. protocols.io. https://dx.doi.org/10.17504/protocols.io.5gig3ue
[24] Michael I Sadowski, Chris Grant, and Tim S Fell. 2016. Harnessing QbD, programming languages, and automation for reproducible biology. *Trends in Biotechnology* 34, 3 (2016), 214–227.
[25] Ashley Sommer and Nicholas Car. 2021. *pySHACL.* https://doi.org/10.5281/zenodo.4750840
[26] Leonid Teytelman, Alexei Stoliartchouk, Lori Kindler, and Bonnie L Hurwitz. 2016. Protocols.io: virtual communities for protocol development and discussion. *PLoS Biology* 14, 8 (2016), e1002538.
[27] Keith F Tipton, Richard N Armstrong, Barbara M Bakker, Amos Bairoch, Athel Cornish-Bowden, Peter J Halling, Jan-Hendrik Hofmeyr, Thomas S Leyh, Carsten Kettner, Frank M Raushel, et al. 2014. Standards for Reporting Enzyme Data: The STRENDA Consortium: What it aims to do and why it should be helpful. *Perspectives in Science* 1, 1-6 (2014), 131–137.
[28] John Vivian, Arjun Arkal Rao, Frank Austin Nothaft, Christopher Ketchum, Joel Armstrong, Adam Novak, Jacob Pfeil, Jake Narkizian, Alden D Deran, Audrey Musselman-Brown, et al. 2017. Toil enables reproducible, open source, big biomedical data analyses. *Nature Biotechnology* 35, 4 (2017), 314.
[29] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. 2013. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic Acids Research* 41, W1 (2013), W557–W561.