

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 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 be exported for execution as either a manual “paper protocol” or Autoprotocol, and is being further developed as an open community effort.

CCS Concepts: • **Applied computing** → **Life and medical sciences**; • **Computing methodologies** → **Knowledge representation and reasoning**.

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 (January 2022), 23 pages. <https://doi.org/TBD>

1 INTRODUCTION

Laboratory protocols are critical to biological research and development. However, protocols are often difficult to communicate or reproduce, given the differences in context, skills, instruments, and other resources between different projects, investigators, and organizations. One of the necessary preconditions for effectively addressing these challenges is for there to be at least one commonly used data representation for describing 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.a.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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2022 Association for Computing Machinery.

Manuscript submitted to ACM

Manuscript submitted to ACM

53 While there has been much prior work on representations for protocols, prior approaches have generally been limited
54 either by their dependence on natural language or in the expressiveness of their representation. Many protocol represen-
55 tations focus on simplifying the capture and distribution of descriptions in natural language, such as protocols.io [27]
56 and the many commercial electronic laboratory notebook products. A similar approach is used for recording protocol
57 execution information with community-defined minimum information standards such as MIAME [7], MIFlowCyt [14],
58 and STRENDA [28]. Much of the key information in such approaches is encoded using natural language, which is
59 easier to solicit from experimentalists, but cannot be readily interpreted by machines. As a consequence, protocols
60 and protocol execution records captured with such representations cannot be automatically validated and are often
61 ambiguous, incorrect, or lacking key information. For the same reasons, such protocols cannot generally be executed by
62 laboratory automation systems.

63
64
65 Other protocol representations have focused specifically on automation-assisted execution. In many cases, these are
66 highly specific solutions tied to specific hardware, often proprietary and tied to particular vendors. Some representations
67 have been made applicable to a broader set of automation systems, however, such as Autoprotocol [17] and Antha [25],
68 or instead use laboratory technicians as their automation, as in the case of Aquarium [12]. All of these representations,
69 however, have been generally “low level” in their description of protocols, focusing on very specific details of each
70 operation. This specificity, which on the one hand enables automated execution, on the other hand poses barriers to
71 adoption and to generalization and reuse, since this level of detail often obscures understanding and is too tied to the
72 specifics of a particular laboratory to readily transfer into different environments. Such representations are typically
73 also difficult to translate into more “human-friendly” forms.

74
75
76 Finally, there are a number of workflow languages that solve similar problems in business logic or information
77 processing, such as UML [19], the Common Workflow Language [2], Taverna [30], and Cromwell [8], not to mention
78 biology-specific workflow systems such as Toil [29] and Galaxy [11]. The execution models of such systems are in some
79 cases are general enough to be applied to the description and execution of laboratory protocols, but to the best of our
80 knowledge such an application has not previously been implemented. Further, their very generality can make it more
81 difficult for a domain expert to see how to apply them to protocols, given the large gap between abstract task execution
82 concepts and the specifics of particular tasks that must be performed in a laboratory.

83
84
85 Here, we present a unified approach to protocol representation that addresses all of these disparate needs and bridges
86 prior approaches through the development of the Protocol Activity Markup Language (PAML), a free and open protocol
87 representation building on a foundation of UML [19], Autoprotocol [17], and SBOL3 RDF [3, 16]. In Section 2, we
88 elaborate on the design goals for PAML, its three foundations, and the approach taken for implementation. Section 3
89 outlines the key elements of PAML’s representation for protocols, libraries of primitive actions, and execution records.
90 Section 4 then discusses the current prototype implementation of PAML and the execution environments it supports,
91 and Section 5 discusses ongoing plans for development.

92 93 94 95 96 97 98 **2 ARCHITECTURE**

99 We begin by elaborating a minimal set of design requirements necessary for any broadly applicable representation
100 of biological protocols. These requirements led us to identify a set of core representational ingredients sufficient to
101 address 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 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:

- **Execution 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. Many laboratories, however, do not have automation available. Moreover, even when some degree of automation is available, it is common for protocols to incorporate both automated and manual stages, so protocols also need to be able 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 time of execution, the laboratory and personnel that executed the protocol, equipment used, etc. A protocol representation thus needs to include support for creating persistent data structures that record a specific protocol execution and linking such execution records back to protocol specifications. It is also important to be able to support automation in metadata tagging of the data collected in the course of protocol execution. For example, given a protocol that collects flow cytometry data on samples of different strains under varying growth conditions, the protocol specification should support automatic association/markup linking each FCS file with information about 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 are the anticipated tolerances 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, to be executed at a lab with different equipment from the lab at which the protocol originated, to enable the protocol to be scaled up or scaled down, etc. A likely use pattern is to have a protocol initially be "too strict" (too specific) to be instantiated in a new lab that wishes to run the protocol, and thus need modification. Rather than creating a new protocol, ideally, the original should be generalized to allow it to run both where it could before and also in the new lab. This improved version could then be contributed back and released as an updated version of the existing protocol. Alternatively, if for some reason generalization is impossible or impractical, users should be able to create a variant and record the source of that variant.
- **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 (e.g., the classic "inadequate methods section" issue in scientific publications).

157 Supporting protocol authors in achieving correctness is thus an important goal for a protocol representation,
158 and while the implementation will depend on specific tooling, the representation specification must provide
159 guidance as to what it means for a protocol to be complete, consistent, etc. This is especially important for
160 automatically-executed protocols since the control system cannot be counted on to repair flaws in protocols on
161 the fly, and in the worst case, an incorrect specification could even cause damage to equipment or endanger lab
162 personnel.
163

164 • **Planning, scheduling, and allocation of laboratory resources:** Laboratory resources are valuable, and
165 some organizations will want to be able to optimize their use. To do so, a protocol representation should support
166 (at least) extraction of resource requirements and estimated durations from activities in the protocol. Note that
167 the specifics of resource requirements and duration estimates will likely be a function of both the protocol and
168 the available equipment in the laboratory in which it is to be executed. Which resources are limited, and must
169 be considered in a planner or scheduler, versus those that can be effectively treated as unlimited, will also vary
170 by laboratory, as will management styles and applicable policies.
171
172
173

174 2.2 Foundations: UML, Autoprotocol, and SBOL RDF 175

176 In developing the PAML protocol representation, we adopted a principle of building upon existing standards wherever
177 possible, in order to increase compatibility and interoperability, take advantage of existing tooling, and make the
178 implementation as lightweight as possible. We introduce these foundations here at a conceptual level, while the specifics
179 of their usage are provided in Section 3.
180
181

182 2.2.1 *UML Behavior Models.* As the core of a protocol is a workflow of activities to be carried out, we began by
183 identifying an established standard for workflow modeling that could provide both a well-defined and general formal
184 semantics, yet also be sufficiently abstract as to allow succinct expression and adaptation. We found such a model in
185 Unified Modeling Language (UML) behavior representations (specifically the current version 2.5.1 [19]). UML behaviors
186 provide a general, domain-independent workflow model. This model encodes a formal execution model based on
187 token-passing, which can support serial, parallel, non-deterministic, and distributed execution. Furthermore, as UML
188 use often focuses on diagram-based communication, it also provides a set of flow control and abstraction constructs for
189 succinct and human-friendly communication about complex workflows. At the same time, the formal execution model
190 provides an unambiguous semantics for verification, validation, and other forms of machine reasoning.
191
192
193

194 2.2.2 *Autoprotocol Laboratory Primitives.* As UML is domain-independent, it does not provide any guidance on what
195 primitive behaviors are suitable for expressing laboratory-independent behaviors. For this, we turn to Autoprotocol [17].
196 Autoprotocol describes biological protocols in terms of a sequence of instructions, and while this linear workflow is not
197 expressive enough for our requirements, the instructions themselves are primitives that can be readily mapped from
198 one laboratory environment to another. These instructions, such as `liquid_handle`, `incubate`, `provision`, and `spin`
199 have been specifically designed and refined by the authors of Autoprotocol as a basis set for expressing the activities of
200 common biological protocols in a manner readily transported between different pieces of laboratory automation. The
201 Autoprotocol instructions thus provide a reasonable starting point for our library of laboratory-independent primitive
202 behaviors.
203
204

205 Some activities expressed using these primitives are at a lower level than desirable for a human experimenter,
206 however, such as specifying pipette mixing as a sequence of repetitive liquid handling operations. In cases such as
207
208

209 these, we will choose to not use Autoprotocol instructions as defined, but replace them with more complex or abstract
210 alternatives that instead capture an Autoprotocol use pattern.
211

212
213 2.2.3 *SBOL 3 Materials, Records, and RDF*. While UML models processes, it does not actually provide a representation
214 to capture executions and associate traces with data. For this, we turn to the Synthetic Biology Open Language (SBOL),
215 version 3 [3, 16], which uses Semantic Web practices and resources, such as *Uniform Resource Identifiers* (URIs) and
216 ontologies, to unambiguously identify and define biological system elements and to provide serialization formats for
217 encoding this information in electronic data files.
218

219 While the early versions of SBOL focused only on genetic designs, it has since been expanded to represent and link
220 information throughout the design-build-test-learn workflow [16]. SBOL provides succinct representations for all of
221 the materials that would be used by a typical biological protocol—strains, reagents, media, experimental sample designs,
222 etc.—along with the ability to track and distinguish between specific physical aliquots and replicates. On the input
223 side of a protocol, SBOL’s combinatorial design specifications [23] offer the ability to compactly specify combinations
224 of experimental conditions. SBOL also incorporates the *Ontology of Units of Measure* (OM) [22] for specifying and
225 recording measurements, as well as the *W3C Provenance Ontology* (PROV-O) [18] for linking specifications, samples,
226 and data via traces of activity records.
227

228 This last is precisely complementary to our selection of UML behaviors for representing workflows, as PROV-O leaves
229 the actual definition of activities to users. Recall that a key objective of PAML is to aid in tracing the connections between
230 data sets and the protocols that produced them—the issue of data set provenance. The provenance ontology (PROV-O)
231 is a well-accepted tool for encoding this kind of information, so we propose to use it to anchor the data-producing
232 relationship between protocols, executions of protocols, and the resulting data sets. PROV-O also specifies several
233 annotation properties that we adopt to mark up protocol executions. Thus, we can use PROV-O as the basis for capturing
234 execution traces, with the activities in the trace defined using the UML data model, built from laboratory primitives
235 based on Autoprotocol, and the inputs, outputs, and data relations encoded using SBOL 3.
236

237 SBOL’s semantic web basis has also been used to allow representational extensions with custom classes without
238 requiring changes to the underlying specification, unlike UML or Autoprotocol. For this reason, we select SBOL RDF
239 as the underlying data model for PAML and convert the relevant portions of UML and Autoprotocol into SBOL RDF
240 extensions. This approach also allows PAML to be extended with additional custom information for particular uses and
241 deployments.
242

243 Critically, SBOL RDF provides a partial closure reasoning model (see Section 5.5 of [3]) that allows much stronger
244 and more “object-oriented” reasoning than plain RDF or OWL, while still allowing documents to reference external
245 material. This allows for an intuitive chunking and linking of information, for example, being able to store and reason
246 about a complete record of a protocol execution that has links to the protocol without being required to store a copy of
247 the protocol in the same document.
248

249 Finally, SBOL RDF also offers a natural approach to management of protocol modifications and versioning, since
250 SBOL RDF can be serialized into the sorted N-triples RDF format. This format is a stable serialization that can be
251 readily differenced and inspected with standard, text-based version control tooling. Thus, implementing PAML using
252 SBOL RDF allows protocols to be maintained by distributed communities of contributors using standard software
253 development version control such as git, as well as the larger ecosystem of associated tooling for project management
254 and community-driven development.
255

261 3 PAML DATA MODEL

262 Following the architecture presented in Section 2, the PAML data model is formulated as an extension of SBOL 3,
263 implemented by encoding an ontology for the UML behavior model and its supporting classes, plus additional classes
264 for linking this information into PROV-O records, libraries of primitive UML activities based on Autoprotocol, and
265 additional classes for tracking laboratory samples and data. In this section, we present all of the major concepts and
266 classes of PAML; additional supporting classes and the complete current specification can be found in the PAML draft
267 specification at <http://bioprotocols.org>.
268
269

270 Following the pattern of SBOL, PAML’s data model is defined in terms of **classes**, instantiated as **objects**. Classes
271 support **inheritance** based on subclass and superclass relationships. Under the SBOL RDF partial closure model, all
272 objects inherit from the `sbol:Identified` type, which supports generic identity and annotation information. Objects
273 that describe a complete subsystem for reasoning additionally inherit from the `sbol:TopLevel` type, and are always
274 bundled with their “child” `sbol:Identified` objects, allowing strong closed-world reasoning over any collection of
275 `sbol:TopLevel` objects. Examples of `sbol:TopLevel` classes in PAML are `paml:Protocol` and `paml:BehaviorExecution`,
276 while examples of child classes are `uml:ActivityNode` and `uml:Parameter`, which are only useful in the context of
277 the `paml:Protocol` they are used to describe.
278
279

280 Classes contain data in the form of **properties**, which may have primitive XML types (string, integer, long, float,
281 boolean, URI, etc.) or may refer to another object via its URI; the former are **data properties**, and the latter **object**
282 **properties**. Diagrams for the data model use UML conventions, in which classes are shown as rectangles labelled at
283 the top with their class name (or containing nothing but the class name for classes defined elsewhere), and an arrow
284 with a hollow triangle at its head represents the inheritance relationship between subclass (tail) and superclass (head).
285 Data properties are shown in text within the rectangle, object properties shown via an arrow with a diamond at the
286 end, and class inheritance by arrows with empty heads. For object properties, filled diamonds indicate **association**
287 **properties**, in which an object “owns” a child object: this is important because it means the child object is bundled with
288 its parent under SBOL RDF’s partial closure model and always moves with its parent in an RDF document to enable
289 strong (closed-world) reasoning about its contents. Empty diamonds, on the other hand, indicate references to objects
290 defined elsewhere, which may or may not be available to reason about without retrieving additional documents. Finally,
291 the number of values a property can have is indicated by upper and lower points on its **cardinality**: [1] (shorthand for
292 1 . . . 1) indicates a required property, [0 . . . 1] indicates an optional property, [0 . . . *] (the empty constraint) indicates
293 a property that can have any number of values, and [1 . . . *] a property that must have at least one value.
294
295
296
297

298 For clarity, since there are several ontologies involved in the implementation of the PAML data model and at least
299 pair of important terms from different ontologies with the same shortened name (`uml:Activity` vs. `prov:Activity`),
300 we include ontology prefixes for all terms in the text. Likewise, in the figures presenting the data we color model classes
301 by ontology and include their ontology prefix; properties are not, however, as they always belong to the ontology of
302 their defining class (e.g., `uml:ownedParameter` is a property for `uml:Behavior`).
303

304 To illustrate the data model, we will use the iGEM LUDOX protocol for calibration of plate reader optical density
305 (OD) measurements [24], first introduced in the 2016 iGEM interlaboratory study [6] and refined thereafter [5]. This is
306 an extremely simple protocol consisting of three steps: water is added to four wells in a 96-well plate, LUDOX silica
307 suspension is added to another four wells, and then all eight samples are measured at some specified absorbance (600nm
308 in its original usage), in order to obtain a baseline measurement of OD, validate machine behavior, and allow path-length
309 correction.
310
311

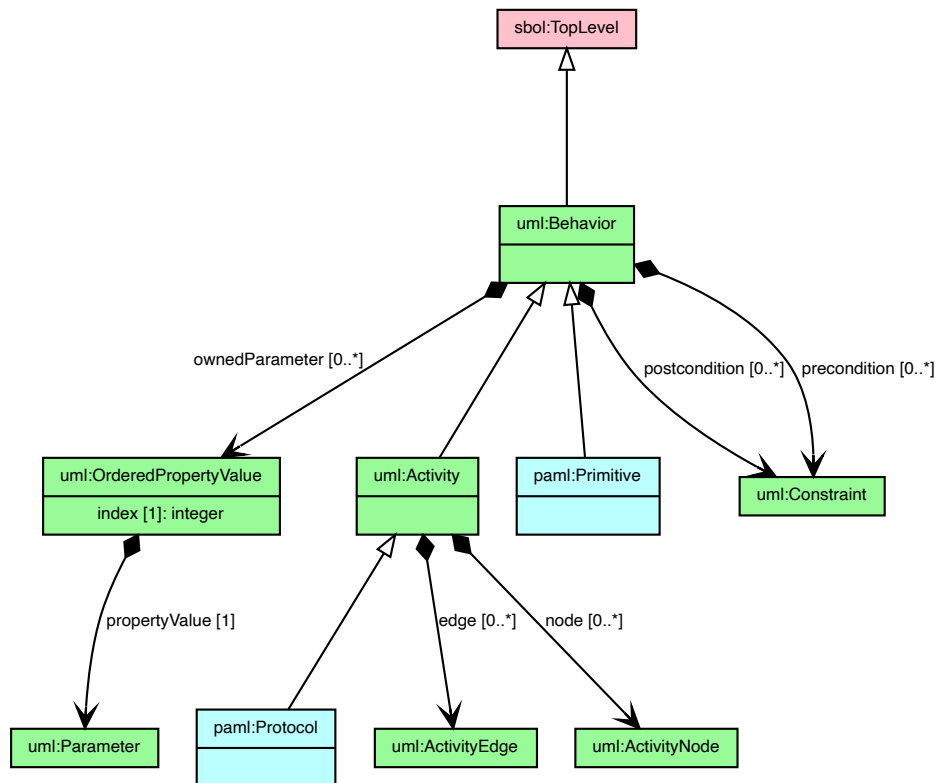


Fig. 1. The uml:Activity and uml:Behavior classes. A paml:Protocol is defined as a uml:Activity, while a paml:Primitive laboratory action is defined as a uml:Behavior.

3.1 Protocols

In UML 2.5.1 [19]), the basic building block of process modeling is a uml:Behavior, which is an abstract specification for how the state of a system changes over time. A uml:Activity is a type of uml: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 we do not need. Note that this does not require any modification of the SBOL standard, as these are implemented as extension classes outside of the restricted SBOL namespace and conform with the SBOL specification guidance on extension via custom classes.

3.1.1 Laboratory Primitives are UML Behaviors. Figure 1 shows the class structure for the adaptation of uml:Behavior and uml:Activity into SBOL RDF. A UML uml:Behavior provides the definition an interface for a process. The uml:ownedParameter property links to an ordered list of uml:Parameter objects (each marked internally with a

365 direction), which describe the order and type of the input arguments that can be given when the `uml:Behavior` is invoked
366 and of the output values that will be returned when the `uml:Behavior` completes its execution. Additional optional
367 `uml:precondition` and `uml:postcondition` properties provide `uml:Constraint` objects that specify requirements
368 on `uml:Parameter` values before and after execution, respectively.
369

370 In PAML, the simplest protocol building block is a `paml:Primitive`, defined as a subclass of UML `uml:Behavior`.
371 Primitives are basic laboratory operations such as pipetting, measuring absorbance in a plate reader, or centrifuging. For
372 example, the iGEM LUDOX calibration protocol uses a `paml:Primitive` named `paml:Provision` to dispense water
373 and LUDOX into a plate and another named `paml:MeasureAbsorbance` to measure the OD values of these samples
374 (both of these are adapted from Autoprotocol as described below in Section 3.1.3).
375

376 Primitives do not provide any information about how to carry out the corresponding real world action (such as
377 pipetting). Instead, they serve as the handoff point between PAML and an execution environment that knows how to
378 actually carry out such primitives in a laboratory, as described further in Section 4.3. However, the *constraints* on these
379 primitives can be used specify what is expected to happen (how the system changes) when the corresponding actions
380 are performed.
381

382 In order to build useful protocols in PAML, we also need libraries of primitives that are simple enough to be readily
383 reused, yet abstract enough to be easily transferred from laboratory to laboratory. As noted above, Autoprotocol [17]
384 already provides a good beginning set of primitive operations, which have already been validated as both reusable and
385 transferrable between different pieces of laboratory equipment. In adapting Autoprotocol, however, PAML adds two
386 key extensions to make protocols more adaptable and reusable: classes for describing collections of samples and classes
387 for organizing primitive operations into libraries.
388
389

390
391 **3.1.2 Sample Collections.** In Autoprotocol it is only possible to address one location at a time, i.e., a single container or
392 a single compartment within a container, such as a well on a plate. This means that any operation on multiple locations
393 must name every location as a fixed value in the definition of the protocol, which in turn means that protocols are often
394 both extremely large (e.g., individually operating on every well of a 96-well plate) and rigid, since locations cannot be
395 supplied as a parameter value or determined dynamically at runtime.
396

397 In common practice, however, protocols are often naturally described in terms of operations on physical or logical
398 collections of samples, such as “Wells A1 to D2 in a standard 96-well plate,” “A 6 by 3 group of 10ml tubes: three replicates
399 each for six conditions,” or “All wells showing green fluorescence >500 MEFL.” Being able to represent such descriptions
400 directly allows representations to be more compact, more intelligible to humans for both authoring and execution, and
401 also more reusable, since they can be communicated as parameters or determined dynamically. PAML thus includes a
402 `paml:SampleCollection` class, as shown in Figure 2(a), that represents organized collections of samples, either as an
403 n-dimensional `paml:SampleArray` (e.g., the wells of a 96-well plate, a sequence of 10 flasks) or as a `paml:SampleMask`
404 that selects a subset of such an array using a mask of Boolean values.
405

406 In particular, A `paml:SampleArray` specifies an n-dimensional rectangular array of samples, all stored in the same
407 type of container. For example, in the iGEM LUDOX calibration protocol, a `paml:SampleArray` is generated by the
408 `paml:EmptyContainer` primitive to allocate a new 96-well plate for use in the protocol. A `paml:SampleArray` might
409 also be used to describe a set of 10 cell cultures growing in 96-well plate wells, or a set of 6 streaked agar plates,
410 or a single 500 mL flask filled with media. For any non-empty location, the contents are in turn described by an
411 SBOL 3 `sbol:Implementation` object that represents a physical sample and which, in turn, can link to an SBOL 3
412 `sbol:Component` object that describes the mixture of materials in that location. Note that this is a logical array, and
413
414
415

417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468

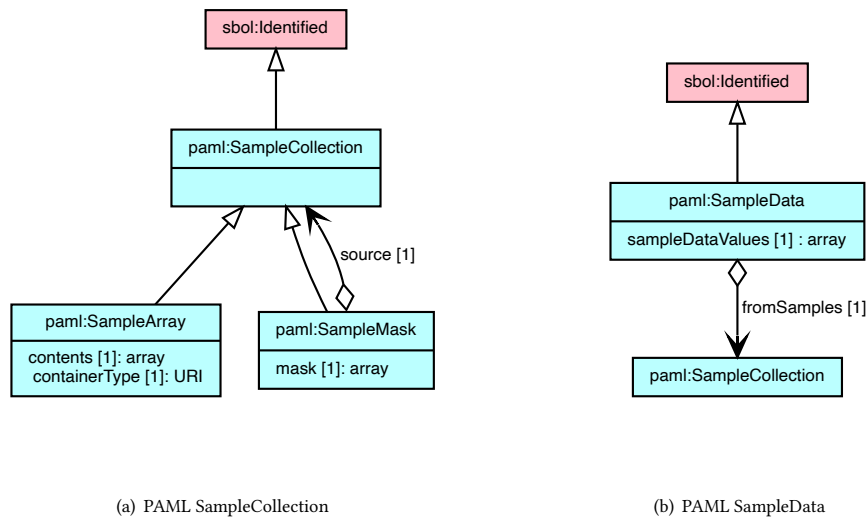


Fig. 2. PAML represents collections of samples as either N-dimensional arrays or using Boolean masks to select subsets from such an array (a). PAML represents sample data by associating an array of data values with a collection of samples (b).

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 sbol:sample array of 96-well plate wells, and then end up laid out over two plates.

As an alternative to a sample array, a paml:SampleMask describes a subset of samples in some paml: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 paml:SampleCollection. In this way, references to the physical laboratory elements on which a paml:SampleArray is laid out can be easily maintained and combined even across different subsetting operations. For example, in the iGEM LUDOX calibration protocol, a paml:SampleMask is used to select wells A1 to D1 on the plate to fill with water and another paml:SampleMask used to select wells A2 to D2 to fill with LUDOX, then a third paml:SampleMask covers both with the range A1 to D2 for measuring absorbance.

Finally, PAML defines another class, paml:SampleData, as shown in Figure 2(b), in order to capture the relationship between physical samples and information about those samples. The paml:SampleData class simply associates a paml: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. The joining of samples and data

PAML Library/Primitive	Autoprotocol Equivalent
Library: sample_arrays	
EmptyContainer	Undocumented “SUPPLY NEW CONTAINER” operation
PlateCoordinates	<i>n/a - Autoprotocol references only single locations</i>
Rows	<i>n/a - Autoprotocol references only single locations</i>
Columns	<i>n/a - Autoprotocol references only single locations</i>
DuplicateCollection	<i>n/a: operation makes an array with the same shape as an input</i>
ReplicateCollection	<i>n/a: like DuplicateCollection, but also adds a new replicate dimension</i>
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
<i>Currently unimplemented</i>	acoustic_transfer, flow_cytometry, measure_mass, measure_volume, spectrophotometry mode=luminescence/shake

Fig. 3. PAML’s primitive laboratory operations are organized into libraries based on required equipment types. The initial collection of “built-in” primitives are based on Autoprotocol, currently implementing most functionality from that language, plus primitives for defining and manipulating collections of samples.

with the `paml:SampleData` class also allows information to be fed back into the computation of `paml:SampleMask` objects at runtime, such as the example above of “all wells showing green fluorescence >500 MEFL.”

In sum, with the addition of the `paml:SampleCollection` and `paml:SampleData` classes, PAML allows primitives to 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. While there are some cases, such as our simple example of the iGEM LUDOX calibration protocol, where specific locations are appropriate, a great many protocols are intended to be able to run over a set of samples or conditions that are provided as input. PAML primitives are thus in general defined for the more general case of collections, with references to specific locations being just one of the ways in which such a collection may be defined.

3.1.3 Primitive Libraries. PAML also adds libraries for organizing collections of primitives. In Autoprotocol there is a fixed set of primitives defined by the specification, which limits extensibility. For PAML, we take an approach like that used in most modern programming languages, in which the specified language is kept as small as possible, while

521 many of the capabilities of the language are provided by collections of functions grouped into libraries, each library
522 associated with a well-defined cluster of functionality.

523 Figure 3 shows how the current implementation of PAML organizes operations taken from Autoprotocol into
524 four libraries of `paml:Primitive` objects. Three of the libraries are classes of laboratory activities that operate on
525 `paml:SampleCollection` objects. The `paml:plate_handling` library contains operations performed on containers
526 (e.g., plates, flasks), such as sealing and incubation, which are mostly direct mapping of equivalent Autoprotocol
527 activities. The `paml:liquid_handling` library contains operations moving liquids within or between containers, such
528 as pipetting from one location to another or using a pipette to mix fluids in a location. This includes a division of the
529 omnibus Autoprotocol `liquid_handle` operation into several different patterns of usage, which we have chosen to
530 do in order to make higher-level abstractions that are both more readily human-interpretable and also more readily
531 accessible for machine reasoning and verification. The third library, `paml:spectrophotometry`, does the same for plate
532 reader measurements. Finally, we have implemented one library that contains mostly new operations (i.e., with no
533 Autoprotocol equivalents) for creating and subsetting `paml:SampleCollection` objects. A few Autoprotocol operations
534 are not included in the current implementation, but are expected to be adapted to add into existing or new libraries as
535 implementation continues.

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

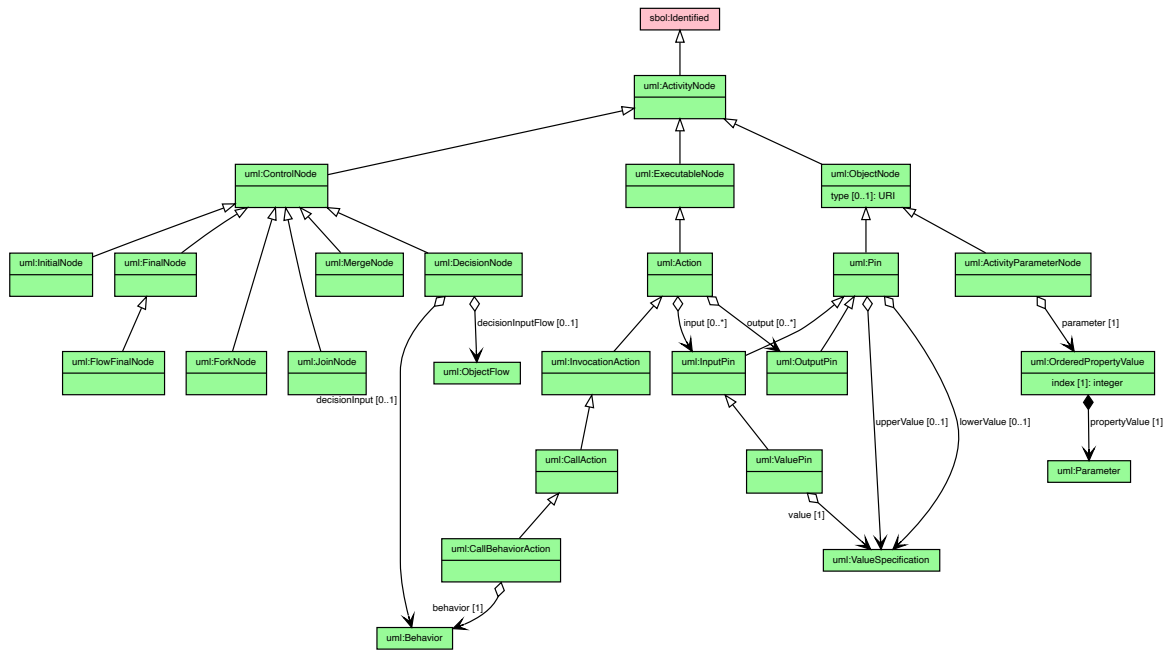
543 **3.1.4 Protocols are UML Activities.** We now move from individual laboratory operations to complete protocols. In UML,
544 a `uml:Activity` is used to define composite behaviors in terms of a network of steps. Its `uml:node` property stores a col-
545 lection of `uml:ActivityNode` objects (Figure 4(a)). In the current implementation of PAML, these `uml:ActivityNodes`
546 can only be of three sub-types:

- 547 (1) A `uml:CallBehaviorAction` object defines an actual step of the protocol, in which a specific `paml:Primitive`
548 or sub-`paml:Protocol` is carried out (note that this class is several layers deep in a hierarchy of other sibling
549 UML `uml:ExecutableNode` classes not currently used in PAML).
- 550 (2) A `uml:ObjectNode` defines a point where data enters and exits the `uml:Activity` (by means of a
551 `uml:ActivityParameterNode`) or one of its `uml:CallBehaviorAction` nodes (by means of a `uml:Pin`).
- 552 (3) A `uml:ControlNode` is used to define where the steps of the `uml:Activity` start (`uml:InitialNode`), stop
553 (`uml:FinalNode`), branch (splitting at a `uml:DecisionNode` and rejoining at a `uml:MergeNode`), or run in
554 parallel (splitting at a `uml:ForkNode` and rejoining at a `uml:JoinNode`).

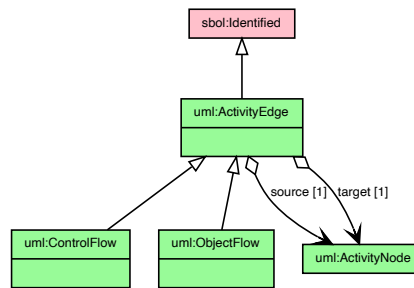
555 The `uml:ActivityEdge` objects that connect pairs of `uml:ActivityNode` objects, on the other hand, are much simpler,
556 merely indicating a path by which either a control or object token flows from a source to a target.

557 The execution semantics for a `uml:Activity` are based on a notion of token flow similar to Petri nets [21]. A Petri
558 net is a graph in which nodes have input edges and output edges. Roughly speaking, a Petri net node is enabled to
559 “fire,” sending a token out all of its output edges, when all of its input edges are filled with tokens; at the same time,
560 those input tokens are removed. When a `uml:Activity` (in this case, the node corresponding to the `paml:Protocol`)
561

573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624



(a) UML ActivityNode



(b) UML ActivityEdge

Fig. 4. A UML `uml:Activity` is defined as a network of `uml:ActivityNode` objects that define steps, decisions, or values in the activity and `uml:ActivityEdge` objects that connect between them.

625 begins execution, tokens start in any `uml:InitialNode` or input `uml:ActivityParameterNode`. From there, they flow
626 along the connected `uml:ActivityEdge` for each such source node to the connected target node. As tokens arrive
627 at a `uml:ActivityNode`, it waits until a token has arrived along every edge for which it is a target; once all edges
628 have delivered a token, the `uml:ActivityNode` takes action if needed (i.e., it is a `uml:CallBehaviorAction`), then
629 sends a token along every edge for which it is a source. The exceptions are `uml:DecisionNode`, whose purpose is
630 to choose one of several edges on which to send a token, and `uml:MergeNode`, its complement whose purpose is to
631 accept a token arriving from one of several edges. A `uml:CallBehaviorAction` node, on the other hand, also waits
632 for incoming tokens to the `uml:InputPin` objects that define values for the input parameters of the `uml:Behavior` it
633 will call and sends tokens from its associated `uml:OutputPin` objects. This process continues, with tokens proceeding
634 along nodes and edges until execution comes to an end as tokens are absorbed by `uml:FinalNode` and/or output
635 `uml:ActivityParameterNode` objects (or, if the execution fails, when no further execution is possible because no node
636 has all of its input edges filled).

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

650 For example, the complete iGEM LUDOX calibration protocol is shown in Figure 5. This `pam1:Protocol` consists
651 of 11 `uml:ActivityNodes` (not counting `uml:Pins`) with 15 `uml:ActivityEdges` connecting the nodes into a net-
652 work. Together, they implement a `uml:Behavior` (in this case, a `pam1:Protocol`) with an interface of one input
653 `uml:Parameter`—the wavelength to be measured—and one output `uml:Parameter`, the absorbance measured at the
654 plates.

656 In this implementation, two nodes initiate the execution of the protocol: the `uml:InitialNode` and the
657 `uml:ActivityParameterNode` for the wavelength input parameter. To manage the ordering of the steps,
658 `uml:ControlFlow` edges are added, the first of which runs from the `uml:InitialNode` to a `uml:CallBehaviorAction`
659 invoking `EmptyContainer` from the `sample_arrays` library. This `pam1:Primitive` requests allocation of a
660 `pam1:SampleArray` for a 96-well clear flat-bottom plate (specified by the `plateRequirement` input not expanded
661 in the visualization). Performing `pam1:EmptyContainer` in turn enables three `uml:CallBehaviorAction` nodes for
662 `PlateCoordinates` operations from the same library, with a `uml:ForkNode` implicitly inserted to support the same
663 plate being accessed multiple times.

666 Note that the `Provision` operations do not produce outputs, instead acting by modifying state of the
667 `pam1:SampleCollection` provided by their destination input. As such, it is necessary to add control edges that
668 ensure that the water is added before the LUDOX, and that both have been put in the plate before absorbance is
669 measured. Finally, once the lab work has been carried out, the final output `uml:ActivityParameterNode` collects and
670 reports the `pam1:SampleData` that is output from the call to `MeasureAbsorbance`.

672 Since a `pam1:Protocol` is itself also a `uml:Behavior`, it can be embedded in other, more complex protocols, including
673 ones with complex hierarchical structures. For example, a `pam1:Protocol` might specify a cell culturing protocol that
674 invokes media adjustment and data collection sub-protocols at various time points, or a multi-stage DNA assembly
675

676

677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728

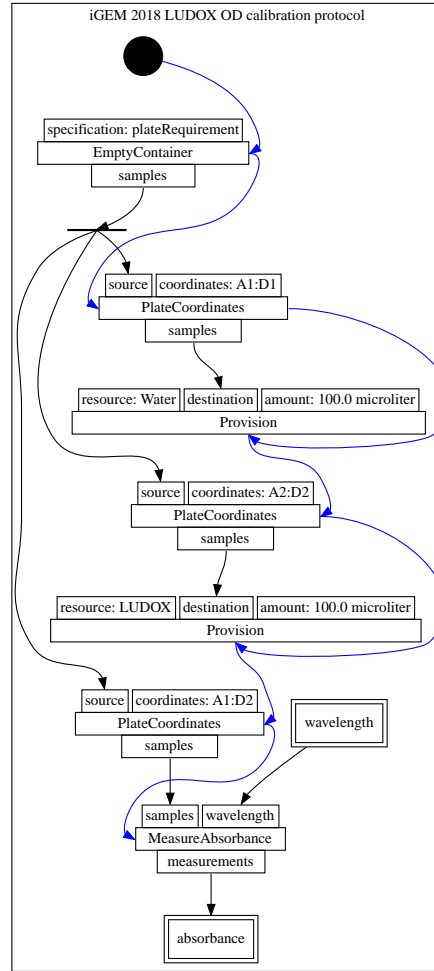


Fig. 5. PAML protocol for the iGEM 2018 LUDOX calibration protocol, automatically rendered by PAML visualizer. The graph includes protocol activities that follow a control flow denoted by blue edges and data flow denoted by black edges. Per UML diagram conventions, the `uml:InitialNode` is shown as a black circle and `uml:ForkNode` is shown as a black bar. The graph also illustrates protocol input (e.g., wavelength) and output (e.g., absorbance) parameters with double boxes.

protocol involving multiple rounds of digestion, ligation, transformation, and selection. Furthermore, requesting the execution of a `paml:Protocol` on a particular set of samples or conditions can itself be represented as a simple “wrapper” `paml:Protocol` in which the desired sample and condition values are supplied as `uml:ValuePin` inputs (a `uml:ValuePin` is a `uml:Pin` with a constant value) on a `uml:CallBehaviorAction` invoking the `paml:Protocol` and its results are collected to output via a `uml:ActivityParameterNode` for each expected result. In principle, even an entire experimental campaign could be encoded in such a manner, if desired.

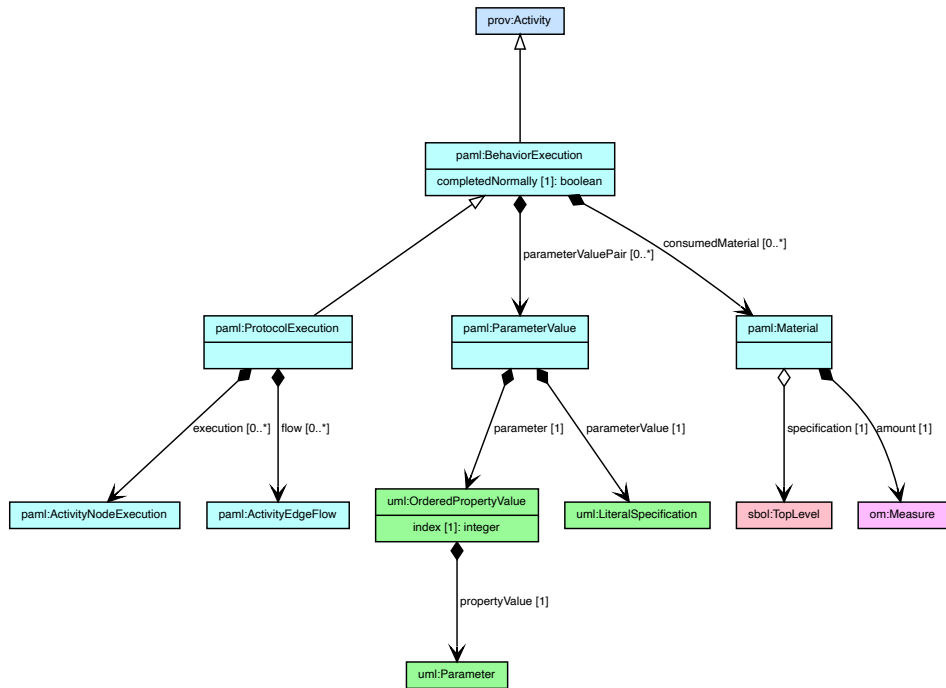


Fig. 6. The `paml:ProtocolExecution` and `paml:BehaviorExecution` classes are used for recording the execution of `paml:Protocols` and `paml:Primitives`.

3.2 Execution Records

PAML's representation for recording the trace of an execution is built on the W3C provenance ontology (PROV-O) [18], which provides a mechanism for recording traces. Using this representation will allow PAML to record how data sets are produced as a result of a behavior execution.

PROV-O has its own distinct notion of a `prov: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:Activity` objects, each linked by its `prov:type` property to a corresponding `uml:Behavior` object specifying the nature of the activity. The `prov:Activity` is only a stub class intended for extension, however. Accordingly, in order to provide additional information needed for recording information about protocol executions, PAML extends `prov:Activity` with a `paml:BehaviorExecution` class for recording execution of a UML `uml:Behavior` (either `paml:Primitive` or `paml:Protocol`) and a child `paml:ProtocolExecution` class for recording further information about the execution of a `uml:Activity` (i.e., `paml:Protocol`), as shown in Figure 6.

A `paml:BehaviorExecution` is a record of how a `paml:Protocol`, `paml:Primitive`, or other `uml:Behavior` was carried out. Such an execution might be either real or simulated, for example as part of "unrolling" a protocol for certain execution environments as described below in Section 4.3. Properties inherited from PROV-O provide the basic structure

781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832

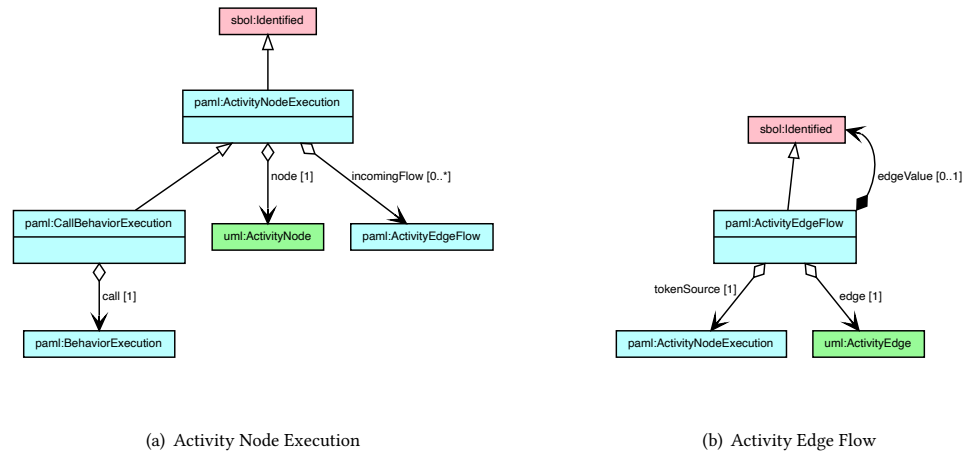


Fig. 7. A UML `uml:Activity` is defined as a network of `uml:ActivityNode` objects that define steps, decisions, or values in the activity and `uml:ActivityEdge` objects that connect between them.

of the trace: the `prov:type` property links to the `uml:Behavior`, its `prov:startedAtTime` and `prov: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:Association` to a `prov:Agent`. For details on usage of PROV-O within an SBOL RDF context, see Appendix A.1 of [3]. Additional PAML-specific properties record the input and output `paml:Parameter` values for the `paml:Behavior`, the laboratory materials consumed (as pairs of an `sbol:Component` specifying material type with an `om:Measure`), and whether the execution completed normally or if there was some exception condition.

A `paml:ProtocolExecution` extends this with the addition of records for the nodes and edges defining the `paml:Protocol`'s behavior as a `uml:Activity`. Specifically, the `paml:execution` property is used to record each firing of a `uml:ActivityNode` and the `paml:flow` property is used to record each time a token moves along a `uml:ActivityEdge`, using the `paml:ActivityNodeExecution` and `paml:ActivityEdgeFlow` classes shown in Figure 7. In the case of invocation of a `paml:Primitive` or `paml:Protocol` via a `uml:CallBehaviorAction`, a corresponding `paml:CallBehaviorExecution` subclass of `paml:ActivityNodeExecution` also provides a link down to the `paml:BehaviorExecution` sub-trace that records the execution of the `paml:Primitive` or `paml: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 no executions on branches not taken or multiple executions 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.

833 4 PROTOTYPE

834 Following the data model presented above, we have constructed a prototype implementation of PAML in terms of
835 an ontology, derived specification document and Python code library, and prototype software tools for visualization,
836 editing, and execution of PAML protocols.
837

838 4.1 Ontology-Based Specification

839 The PAML data representation is implemented first and foremost as an ontology encoded in the Web Ontology Language
840 (OWL). We leverage this machine-readable specification to make the standards development process more efficient
841 through automation. First, we automatically generate graphical visualizations of the data model which illustrate the
842 classes, properties, and links between classes. Second, we automatically generate the data model portions of the PAML
843 specification document as LaTeX. This human-readable document incorporates the class diagrams as well as explanatory
844 descriptions which are sourced from annotations contained in the original ontology file, plus manually generated
845 preamble material introducing the motivation and context of the specification. We also automatically generate an
846 object-oriented Python API which supports authoring and exchange of PAML protocols as any of a number of standard
847 RDF formats. For this purpose, we use a tool called SBOLFactory [4] to dynamically generate Python classes directly
848 from the ontology file.
849

850 We use ontology-based specification for several reasons. First, because the standard is still in an early stage of
851 development and is expected to evolve, revisions to the proposed data model can be rapidly generated, released, and
852 tested in practice. Moreover, since the human-readable specification document and the software library are generated
853 from a single source, we avoid introducing errors and discrepancies between the different artifacts. Overall, these
854 factors enable rapid development and responsiveness of PAML developers to emerging needs and use cases from the
855 open protocols community.
856

857 The Python API also performs validation on protocols to ensure that representations are complete and consistent.
858 For example, one such validation rule requires that every ProtocolExecution must link to a Protocol. We use the Shapes
859 Constraint Language (SHACL), an RDF-based language that describes graph patterns to which valid data instances
860 must conform [13], to encode these validation rules in a declarative syntax. The pySHACL validator tool [26] can then
861 be used to check conformance of PAML objects in an RDF document according to the encoded rules. By using this
862 combination of OWL and SHACL, we can fully specify the PAML data model using non-ambiguous, machine-readable
863 languages. The specification is thus decoupled from its implementation in any one programming language as well as its
864 formulation in natural language.
865

866 4.2 Visualization and Editing of Protocols

867 Graphs are a natural model for visualization and editing of PAML protocols because of their basis in the UML Activity
868 model, in which each protocol involves a set of activities and controls (nodes) that are linked by data and control
869 flows (edges). Figure 5 illustrates the rendering of the iGEM 2018 LUDOX calibration protocol via GraphViz [10]. It
870 includes an initial control node (filled black circle) that is followed by (denoted by a blue control edge) an activity
871 node EmptyContainer. The activity nodes include input pins (e.g., specification) and output pins (e.g., samples).
872 Data flow edges (denoted by black edges) link the activity pins (e.g., EmptyContainer output samples pin links to the
873 source input pin for PlateCoordinates) either directly or through control nodes such as a fork node (denoted by a
874 black bar). Data flow edges also link protocol input (e.g., wavelength) and output (e.g., absorbance) parameters.
875

885 While Figure 5 illustrates the protocol as a graph, PAML can be illustrated in a number of formats. The protocol
886 illustrated by Figure 5 may also be described as a list of activities because the activities are totally ordered. PAML
887 protocols that are partially ordered or include decision nodes can be represented by other paradigms such as block-based
888 programs [15] or visual scripts [1, 9]. More broadly, protocols can be viewed as programs that can be expressed in a
889 programming language or pseudocode.
890

891 In addition to visualizing protocols, the same paradigms support editing, such as adding, deleting, or configuring
892 activities. The PAML Python API supports protocol editing operations by providing functions that build a protocol.
893 Protocol generation scripts (e.g., as listed in Figure 8) execute a sequence of API functions that construct the PAML
894 for the iGEM LUDOX calibration protocol. For example, in that script lines 5 to 7 define the protocol and add it to an
895 SBOL3 document (representing the protocol as RDF). Line 10 defines an input parameter called wavelength. Line 13
896 defines the SBOL3 object for double-distilled water, grounding it in a link to a PubChem identifier. Line 18 defines a
897 microplate object that will hold the samples. Lines 21 to 24 identify the wells that will hold water and provision the
898 water into those wells. Lines 29 to 32 identify which wells to measure and then measure the absorbance. Finally, lines
899 35 to 38 define the protocol output parameter for absorbance and link it to the output of the absorbance measurement
900 activity. Visual editors can use these functions to implement the same functionality as the Python script.
901

902 Visualizing protocols, like visualizing source code, helps to specify and understand the protocol. However, like
903 programs, executing a protocol requires interpreting the steps. As the number of objects and control flow nodes increases,
904 it becomes increasingly difficult to interpret the protocol. Furthermore, PAML protocol serialization to different target
905 languages may require compiling away some of the control structure: for example, in Autoprotocol control flow is
906 a strict linear step order with no branching, so a PAML protocol must be linearized with all loops “unrolled” and all
907 sub-protocol invocations expanded inline in order to produce a linearized version of the protocol suitable for mapping
908 into Autoprotocol. Accordingly, in order to support execution either directly or through serialization, we have developed
909 an execution engine for PAML.
910

911 4.3 Execution in Markdown and Autoprotocol

912 PAML execution requires interpreting a protocol to determine which activity can be executed next, and then recording
913 the data generated on the output pins. The PAML execution engine uses a token based execution semantics that
914 implements the UML activity model based upon Petri-nets [21]. The execution involves tracking a set of tokens
915 generated by each activity or control node in the protocol. Each activity generates tokens on their output pins and
916 consumes tokens on the input pins. The tokens can either hold data representing objects created by activities, or can
917 denote control. The execution engine non-deterministically selects an enabled protocol node each iteration. It records
918 the execution for each node in an execution trace and notifies any listeners. Executing a protocol offline (e.g., for
919 export to Autoprotocol) involves generating identifiers for activity outputs that will be specified later, during an actual
920 execution of the protocol. Online execution of the protocol provides an opportunity to specify actual outputs as the
921 activities execute. Figure 9 shows an example visualizing an execution trace for the LUDOX protocol.
922

923 In order to generate serializations of protocols to alternative target formats, the execution engine uses listeners that
924 output either Autoprotocol or Markdown. Both Autoprotocol (a machine language) and Markdown (a list of steps in
925 semi-structured natural language) require a sequence of steps that totally order the protocol activities and omit control
926 flow that is otherwise implicit in the format (e.g., omitting the object fork node for the plate, as illustrated by the black
927 bar in Figure 5). The execution listener pattern is particularly helpful for serializing protocols that include multiple
928

```
937 1 import sbol3
938 2 import paml
939 3
940 4 # declare a protocol and add it to an SBOL3 document doc (doc initialization omitted).
941 5 protocol = paml.Protocol('iGEM_LUDOX_OD_calibration_2018')
942 6 protocol.name = "iGEM 2018 LUDOX OD calibration protocol"
943 7 doc.add(protocol)
944 8
945 9 # add an optional parameter for specifying the wavelength
946 10 wavelength_param = protocol.input_value('wavelength', sbol3.OM_MEASURE, optional=True, default_value=sbol3.Measure(600,
947 11 tyto.OM.nanometer))
948 12
949 13 # create the materials to be provisioned (Ludox omitted for brevity)
950 14 ddh2o = sbol3.Component('ddH2O', 'https://identifiers.org/pubchem.substance:24901740')
951 15 ddh2o.name = 'Water'
952 16 doc.add(ddh2o)
953 17
954 18 # get a plate (spec omitted for brevity)
955 19 plate = protocol.primitive_step('EmptyContainer', specification=spec)
956 20
957 21 # identify wells to use
958 22 c_ddh2o = protocol.primitive_step('PlateCoordinates', source=plate.output_pin('samples'), coordinates='A1:D1')
959 23
960 24 # put water in selected wells
961 25 provision_ddh2o = protocol.primitive_step('Provision', resource=ddh2o, destination=c_ddh2o.output_pin('samples'), amount=
962 26 sbol3.Measure(100, tyto.OM.microliter))
963 27
964 28 # similar Ludox PlateCoordinates and Provision steps omitted
965 29
966 30 # identify wells to use
967 31 c_measure = protocol.primitive_step('PlateCoordinates', source=plate.output_pin('samples'), coordinates='A1:D2')
968 32
969 33 # measure the absorbance
970 34 measure = protocol.primitive_step('MeasureAbsorbance', samples=c_measure.output_pin('samples'))
971 35
972 36 # link input parameter to measure primitive input
973 37 protocol.use_value(wavelength_param, measure.input_pin('wavelength'))
974 38
975 39 # link measurement output to protocol output
976 40 output = protocol.designate_output('absorbance', sbol3.OM_MEASURE, measure.output_pin('measurements'))
977 41
978 42
979 43
980 44
981 45
982 46
983 47
984 48
985 49
986 50
987 51
988 52
```

Fig. 8. PAML Python script to construct a portion of the iGEM LUDOX calibration protocol. An interactive Jupyter notebook is available at: <https://colab.research.google.com/drive/1WpvQ0REjHMEsginxXMj1ewqfFHZqSyM8?usp=sharing>

repetitions of sub-protocols as activities. This requires the protocol executor to “unroll” the protocol into a series of distinct executions, in which sub-protocols may appear multiple times as they are repeated.

PAML is converted to Autoprotocol and Markdown with different execution listeners. For example, the iGEM LUDOX calibration protocol is shown converted and rendered into Markdown in Figure 10 and into Autoprotocol in Figure 11. The listeners not only collect the translated sequence of protocol activities, but also help to resolve the objects appearing the protocol. For example, the Autoprotocol listener interprets the EmptyContainer activity to identify an available container (c.f., line 38 in Figure 11) in the Strateos laboratory information management systems (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, as well as embedding links to definitions, for example in this case linking each material to its NCBI PubChem substance definition, which in turn provides supplier information for the required materials. In addition to interpreting the steps, the listeners also 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.

989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040

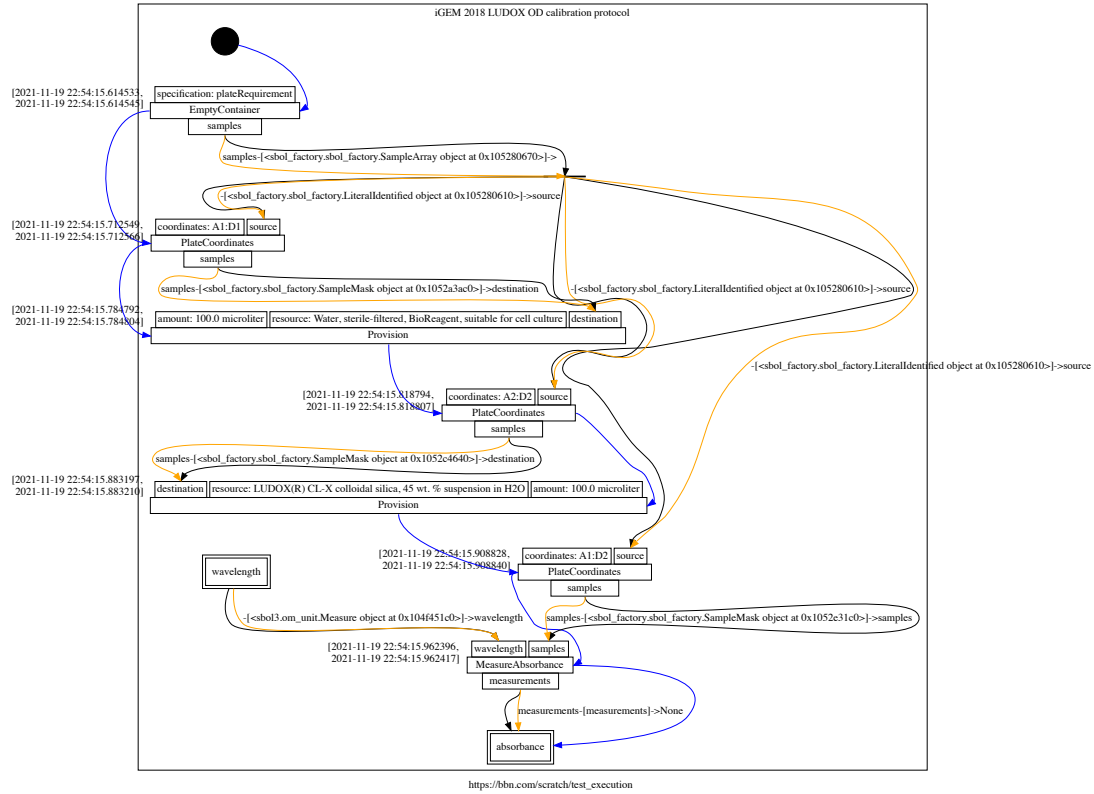


Fig. 9. PAML execution trace for the iGEM 2018 LUDOX calibration protocol, layered on the protocol visualization shown in Figure 5. Yellow edges denote data flow with placeholder and computed values for an offline execution of the protocol.

5 FUTURE DIRECTIONS

The development efforts on PAML described above have produced a draft representation that appears to be simultaneously expressive enough and compact enough to 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, 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 prioritized by this community, are thus:

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](#)
- [LUDOX\(R\) CL-X colloidal silica, 45 wt. % suspension in H2O](#)

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](#) into `samples(A1:D1)`.
 3. Pipette 100.0 microliter of [LUDOX\(R\) CL-X colloidal silica, 45 wt. % suspension in H2O](#) 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. Markdown “paper protocol” generated from PAML for iGEM 2018 LUDOX calibration protocol.

- putting PAML to use in ongoing interlaboratory collaborations within the stakeholder community,
- implementation of additional key execution environments, such as the OpenTrons API [20] and protocols.io [27],
- determining representational details for sample arrays and sample data,
- implementing reasoning about the contents of samples, and
- 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.

```
1093 1 {"instructions": [  
1094 2   { "op": "provision",  
1095 3     "resource_id": "rs1c7pg8qs22dt",  
1096 4     "measurement_mode": "volume",  
1097 5     "to": [  
1098 6       {  
1099 7         "well": "samples/0",  
1100 8         "volume": "100:microliter"  
1101 9       },  
1102 10      <Others omitted>  
1103 11     ]},  
1104 12   { "op": "provision",  
1105 13     "resource_id": "rs1b6z2vgatkq7",  
1106 14     "measurement_mode": "volume",  
1107 15     "to": [  
1108 16       {  
1109 17         "well": "samples/1",  
1110 18         "volume": "100:microliter"  
1111 19       },  
1112 20      <Others omitted>  
1113 21     ]},  
1114 22   { "op": "spectrophotometry",  
1115 23     "dataref": "measurements",  
1116 24     "object": "samples",  
1117 25     "groups": [  
1118 26       {  
1119 27         "mode": "absorbance",  
1120 28         "mode_params": {  
1121 29           "wells": [  
1122 30             "samples/0",  
1123 31             <Others omitted>  
1124 32           ],  
1125 33           "wavelength": [  
1126 34             "600:nanometer"  
1127 35           ]}]},  
1128 36     "refs": {  
1129 37       "samples": {  
1130 38         "id": "ctlg9qsg4wx6gcj",  
1131 39         "discard": true  
1132 40       }  
1133 41     }  
1134 42   }  
1135 43 ]}]}
```

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

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. Approved for Public Release, Distribution Unlimited.

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 Leeher, 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, Chris Myers, Bryan Bartley, Jacob Beal, Matthew Crowther, Thomas E. Gorochowski, Raik Grunberg, Goksel Misirli, Thomas Mitchell, Ernst Oberortner, James Scott-Brown, , and Anil Wipat. 2021. Synthetic biology open language (SBOL) version 3.0.1. <https://github.com/SynBioDex/SBOL-specification/releases/tag/v3.0.1>.
- [4] Bryan Bartley. 2021. SBOLFactory: Ontology-driven code generation. In *HARMONY 2021*.

- 1145 [5] Jacob Beal, Geoff S Baldwin, Natalie G Farny, Markus Gershater, Traci Haddock-Angelli, Russell Buckley-Taylor, Ari Dwijayanti, Daisuke Kiga,
1146 Meagan Lizarazo, John Marken, et al. 2021. Comparative analysis of three studies measuring fluorescence from engineered bacterial genetic
1147 constructs. *PLoS one* 16, 6 (2021), e0252263.
- 1148 [6] Jacob Beal, Traci Haddock-Angelli, Geoff Baldwin, Markus Gershater, Ari Dwijayanti, Marko Storch, Kim De Mora, Meagan Lizarazo, Randy
1149 Rettberg, and with the iGEM Interlab Study Contributors. 2018. Quantification of bacterial fluorescence using independent calibrants. *PLoS one* 13, 6
1150 (2018), e0199432.
- 1151 [7] Alvis Brazma, Pascal Hingamp, John Quackenbush, Gavin Sherlock, Paul Spellman, Chris Stoeckert, John Aach, Wilhelm Ansorge, Catherine A
1152 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.
- 1153 [8] Broad Institute. 2019. *The Workflow Description Language and Cromwell*. <https://software.broadinstitute.org/wdl>
- 1154 [9] Zdena Dobesova. 2011. Visual programming language in geographic information systems. In *Proceedings of the 2nd international conference on*
1155 *Applied informatics and computing theory*. World Scientific and Engineering Academy and Society (WSEAS), 276–280.
- 1156 [10] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. 2003. Graphviz and dynagraph: static and dynamic
1157 graph drawing tools. In *GRAPH DRAWING SOFTWARE*. Springer-Verlag, 127–148.
- 1158 [11] Jeremy Goecks, Anton Nekrutenko, and James Taylor. 2010. Galaxy: a comprehensive approach for supporting accessible, reproducible, and
1159 transparent computational research in the life sciences. *Genome Biology* 11, 8 (2010), R86.
- 1160 [12] Ben Keller, Justin Vrana, Abraham Miller, Garrett Newman, and Eric Klavins. 2019. Aquarium: The laboratory operating system version 2.6.0. (2019).
1161 <https://doi.org/10.5281/zenodo.2583232>
- 1162 [13] Holger Knublauch and Dimitris Kontokostas. 2017. Shapes Constraint Language (SHACL). <https://www.w3.org/TR/shacl/>.
- 1163 [14] Jamie A Lee, Josef Spidlen, Keith Boyce, Jennifer Cai, Nicholas Crosbie, Mark Dalphin, Jeff Furlong, Maura Gasparetto, Michael Goldberg, Elizabeth M
1164 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.
- 1165 [15] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment.
1166 *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15.
- 1167 [16] James Alastair McLaughlin, Jacob Beal, Göksel Mısırlı, Raik Grünberg, Bryan A Bartley, James Scott-Brown, Prashant Vaidyanathan, Pedro
1168 Fontanarrosa, Ernst Oberortner, Anil Wipat, et al. 2020. The Synthetic Biology Open Language (SBOL) version 3: simplified data exchange for
1169 bioengineering. *Frontiers in Bioengineering and Biotechnology* 8 (2020), 1009.
- 1170 [17] Ben Miles and Peter L Lee. 2018. Achieving reproducibility and closed-loop automation in biological experimentation with an IoT-enabled Lab of
1171 the future. *SLAS Technology* 23, 5 (2018), 432–439.
- 1172 [18] Paolo Missier, Khalid Belhajjame, and James Cheney. 2013. The W3C PROV family of specifications for modelling provenance metadata. In
1173 *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 773–776.
- 1174 [19] Object Management Group. 2017. OMG Unified Modeling Language (OMG UML) Version 2.5.1. <https://www.omg.org/spec/UML/>.
- 1175 [20] Opentrons. 2020. OT-2 Python Protocol API Version 2. <https://docs.opentrons.com/v2/>
- 1176 [21] Carl Adam Petri. 1966. *Communication with automata*. Ph.D. Dissertation. Universität Hamburg.
- 1177 [22] 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>.
- 1178 [23] Nicholas Roehner, Bryan Bartley, Jacob Beal, James McLaughlin, Matthew Pocock, Michael Zhang, Zach Zundel, and Chris J Myers. 2019. Specifying
1179 combinatorial designs with the synthetic biology open language (sbol). *ACS synthetic biology* 8, 7 (2019), 1519–1523.
- 1180 [24] Paul Rutten, Richard Tennant, Jacob Beal, Traci Haddock-Angelli, Natalie Farny, Geoffrey Baldwin, Marko Storch, and Ari Dwijayanti. 2019.
1181 Calibration Protocol - OD600 Inter-equipment Conversion with LUDOX. protocols.io. <https://dx.doi.org/10.17504/protocols.io.5gig3ue>
- 1182 [25] 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.
- 1183 [26] Ashley Sommer and Nicholas Car. 2021. *pySHACL*. <https://doi.org/10.5281/zenodo.4750840>
- 1184 [27] Leonid Teytelman, Alexei Stoliartchouk, Lori Kindler, and Bonnie L Hurwitz. 2016. Protocols.io: virtual communities for protocol development and
1185 discussion. *PLoS Biology* 14, 8 (2016), e1002538.
- 1186 [28] Keith F Tipton, Richard N Armstrong, Barbara M Bakker, Amos Bairoch, Athel Cornish-Bowden, Peter J Halling, Jan-Hendrik Hofmeyr, Thomas S
1187 Leyh, Carsten Kettner, Frank M Raushel, et al. 2014. Standards for Reporting Enzyme Data: The STREND Consortium: What it aims to do and why
1188 it should be helpful. *Perspectives in Science* 1, 1–6 (2014), 131–137.
- 1189 [29] John Vivian, Arjun Arkal Rao, Frank Austin Nothhaft, Christopher Ketchum, Joel Armstrong, Adam Novak, Jacob Pfeil, Jake Narkizian, Alden D
1190 Deran, Audrey Musselman-Brown, et al. 2017. Toil enables reproducible, open source, big biomedical data analyses. *Nature Biotechnology* 35, 4
1191 (2017), 314.
- 1192 [30] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra
1193 Nenadic, Paul Fisher, et al. 2013. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the
1194 cloud. *Nucleic Acids Research* 41, W1 (2013), W557–W561.