nature synthesis

Review article

https://doi.org/10.1038/s44160-024-00649-8

Reproducibility in automated chemistry laboratories using computer science abstractions

Received: 25 April 2024

Accepted: 16 August 2024

Published online: 10 October 2024

Check for updates

Richard B. Canty & Milad Abolhasani

While abstraction is critical for the transferability of automated laboratory science in (bio)chemical and materials sciences, its improper implementation is a technical debt taken against the reproducibility of experimental results. Over the decades, computer science has developed guidelines and strategies for how abstractions are captured in programming languages—particularly concerning the substitutability of implementations of abstracted ideas and the clear definition of the contexts in which abstractions are used. However, few programming languages developed for automated experiments fully leverage the wisdom learned in computer science. To achieve collaborative sharing of scientific knowledge via automated laboratories, the way that experimental protocols are codified and interpreted by machine agents must use abstractions responsibly and with reproducibility, rather than solely transferability, at its core. This Review discusses how computer science principles of abstraction can be translated to create more reproducible automation as an enabler for the acceleration of collaborative research with self-driving laboratories.

Reproducibility acts as a proxy measurement of truth and is central to the scientific process. As experiments become more automated, the standardization of process specification and execution is crucial for enhancing experimental interoperability and reproducibility in (bio)chemical and materials sciences. Inspired by the success of cross-platform computer software, many efforts have been made to codify laboratory processes such as Autoprotocol[1,2], χDL[3], the Emerald Cloud Lab Symbolic Lab Language[4,5], AnIML[6], the MAOS language[7] and LabVIEW[8], among others[9–12]. The creators of these programming languages for laboratory automation have taken diverse approaches for addressing how someone (or something) reading these languages can properly recreate an experiment without necessarily requiring an exact copy of the original experimental set-up. These programming languages are tools for recording and prescribing experiments, yet they exist within ecosystems of other tools for writing and comprehending the language. The ways in which these tools interact can have great consequences in the ultimate reproducibility of a codified experiment in (bio)chemistry or materials science laboratories. This Review Article will focus mainly on the qualities of programming languages for laboratory automation in (bio)chemical and materials sciences to address how the abstraction permitted by the language can influence both the reproducibility and transferability of the experiments between laboratories with different experimental set-ups.

Abstraction is a powerful tool in specifying processes—for example, a synthesis procedure can be templated and repeated for each ligand in a library, the actuation of each motor in a robotic arm can be summarily specified with just the pick-up and drop-off locations. By deferring specification or hiding unnecessary information, abstraction reduces complexity and enables greater expressive power and transferability—traits which have made abstraction ubiquitous in computer science. Recent efforts into codifying experimental procedures in organic chemistry[7,13,14] have demonstrated the power of portability whereby the same high-level (abstracted) 'code' is executed using different hardware. In general, the approach of abstracting related implementations on the basis of the high-level objective that they accomplish is a promising strategy to enable collaborating laboratories

Department of Chemical and Biomolecular Engineering, North Carolina State University, Raleigh, NC, USA. e-mail: abolhasani@ncsu.edu

https://doi.org/10.1038/s44160-024-00649-8

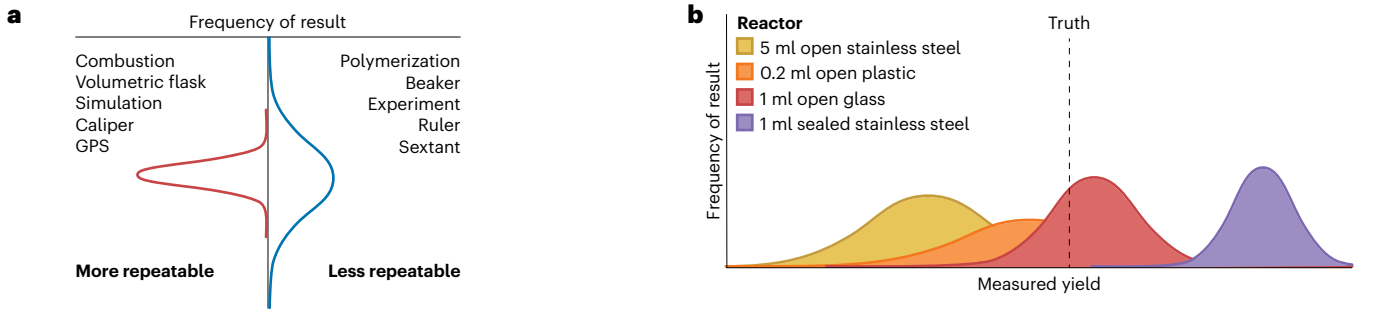


Fig. 1 | Comparisons of the repeatability and reproducibility of experimental results. a, Visual examination of repeatability as controlled by instrumental precision and the inherent stochasticity of the system. GPS, global positioning system navigation. **b**, Demonstration of the validity of procedural changes on the reproducibility of experimental results. In this example, the reaction yield is more reproducible under minor volume changes (0.2–5.0 ml) and the material of the reactor (metal, plastic or glass) than it is under atmospheric control (open versus sealed).

to share and validate experimental workflows[15]. As it pertains to experimental reproducibility, the level of abstraction represents the degree to which modifications to an experiment are permitted to achieve interoperability (for example, vague instructions to remove copper from a solution could be implemented via a colorimetric titration, ion detectors or a precipitating reagent; conversely, a specific instruction to use spin coating rules out drop casting or sputtering). The misuse of abstraction[16], therefore, can result in irreproducibility for experiments—thereby hindering collective scientific advancements.

As the modification of experimental workflows is a requirement for transferring experimental protocols between different laboratories, with acknowledgement of the diverse interpretations of reproducibility, repeatability, replicability and so on[17–19], the following discussions of reproducibility and repeatability will be centred around the distributions of experimental results—with repeatability relating to the variance of a single distribution of results and reproducibility relating to the overlap between multiple distributions of results. Repeatability (Fig. 1a), which applies when the experimental set-up is held constant, is the lack of variance in measurements as determined by the innate uncertainties of the experimental subject and the precision of the experimental methods (for example, the droplet-to-droplet variability of yield and the selectivity in a given flow reactor). Reproducibility is the confidence of agreement between result distributions and is subject to changes in the experiment (Fig. 1b; for example, the variation in yield between different reactors). When experimental modifications are minimized, reproducibility provides a measure of how well an experiment is specified and reported. When experimental modification is permitted, as in the case of abstraction, reproducibility provides a measure of the validity of these abstractions.

Evaluating repeatability and reproducibility is not absolute. Experiments may have multiple goals, and different projects may focus on different objectives. As a result, metrics for repeatability and reproducibility are goal-dependent and often multi-objective. For example, it may be possible to achieve reproducible product yields but not product distributions[20], or to achieve repeatable support for a reaction mechanism but not reproducible kinetic parameters. Similarly, it is possible to reproducibly arrive at a final chemical or material yet have inconsistent intermediates.

Reproducibility is often difficult to predict during development and requires collaborators with similar experimental capabilities for it to be properly measured. Proxy measurements (such as replicate experiments interspaced with random experiments[21,22]) are often used to enable a single laboratory to estimate reproducibility. Evaluating whether or not overabstraction has resulted in a loss of reproducibility requires exactness in which observables are under scrutiny and which statistical techniques are used. Ultimately, abstraction should be leveraged for transferability without adversely impacting reproducibility, and should synergize with automation to produce both precise (repeatable and reproducible) and accurate (close to the ground truth) experimental results.

Languages for the programming and automation of experiments are tailored towards specific objectives, philosophies and applications. Practical considerations can include having the language be easily interpreted by a human (the user experience) or forbidding dangerous actions (security and safety). For experimentalists, reproducibility is paramount because confirming or refuting results sits at the heart of scientific study and the pursuit of knowledge. Failures of reproducibility in automated experimental systems can result in a number of problems, which decelerate research. Failures to reproduce, especially when the causes are hidden behind layers of abstraction[16], can create difficulties with troubleshooting, prolong experimental campaigns with redundant experiments (wasting time and resources) and deter the use and adoption of the technology to accelerate science. Whereas discrepancies can result in discoveries, edge cases and failures of the experimental automation can mislead scientists—hiding discoveries or leading to doomed studies. Crucially, failures to reproduce results or quickly resolve discrepancies (such as the dispute in results produced by the A-Lab of Lawrence Berkeley National Laboratory)[23] erodes trust and support in automated (and autonomous) laboratory ecosystems[24]. The proper codification of experimental workflows into automatically executable codes should enable a collaborative future for research where experimental results can be reproduced with minimal troubleshooting, where automated agents can participate in writing and adapting experiments and where the translation between set-ups (such as between batch and flow chemistry formats) can be automated[25]. Without expressive (Box 1) and interpretable languages that properly encode the reproduction of fundamental operations (such as material transfers, reactions, separations and characterizations), such a collaborative and interoperable future of research will not be achievable (Fig. 2).

In the following sections, we will discuss what abstraction can mean in an experimental context centred around how abstraction implies substitutability in implementations and how these implementations can be given a formalized interface to facilitate transferability. Following these definitions, approaches for responsibly using abstraction seen in computer science are translated to an experimentalist context. Generally, these approaches involve (1) tailoring the programming language to support communication with supplemental processes which monitor experiments and (2) limiting the use of abstraction to cases where implementations are truly interchangeable in practice. Finally, we look to the future in how combinations of these approaches may be leveraged to create (potentially multiple) programming languages for experimental specification that can deliver on the promise for interoperable and reproducible experiments and what that would mean for massively collaborative scientific advances in (bio)chemistry and materials science.

https://doi.org/10.1038/s44160-024-00649-8

BOX 1

Definition of key terms

Expressive power of a language. The gamut of ideas that can be conveyed in a language. Often the definition is restricted to what is easily conveyed[96,97]. For example, a language may be able to specify distillations and reactions but not a reactive distillation.

Abstraction. A simplified but sufficient representation or model that can be used for design.

Interface. The allowed interactions with an abstraction, often specifying what can be asked of the abstraction and how it will reply. An abstraction with a well-defined interface is known as a 'type'; an 'object' is a concrete instance of a type (for reference, these are the 'objects' in 'object-oriented programming'). The interface for an entire software package is commonly known as an application programming interface (API).

Contract. The formal definition of an interaction specified in an interface: what information must be provided, what actions may be taken and how the result will be communicated[98].

Caller. The person, software or process that requests an action—the 'contractee' in the contract analogy.

State. The specific configuration and properties of a material, machine, program or workflow.

Coupling. A measure of how interdependent systems are. In the context of creating good abstractions, this will be the extent to which an abstraction depends on other abstractions' implementations.

Design pattern. A heuristic or best-practice for addressing a common problem—often developed by identifying commonalities across expert solutions to related problems. The design patterns referenced in this work are detailed further in Supplementary Section 1.

Imperative versus declarative programming. The extent to which the procedure or control of a program is made explicit (imperative) or implicit (declarative)[99].

Prescriptive versus descriptive language. The extent to which a language can specify instructions (prescriptive) or detail observations (descriptive). Prescription may use instructions to blindly control latent or hidden parameters; description, however, is confined to only sensible or observable properties.

Compilers, interpreters and transpilers. Software that translates instructions between languages (often between a human-readable language and a machine-readable one) so that a machine can execute the instructions.

Abstraction

Abstraction is a means by which processes and ideas are organized and simplified by focusing on common qualities and behaviours[26], through hiding inconsequential differences[27] or by deferring specification. In chemistry, materials are often referred to in the abstract (reactant, solvent or catalyst rather than specific chemicals), as are processes (workup and separation rather than neutralization and filtration). An abstraction can focus on what a material or process is, what it does, how it is best represented or combinations thereof. There are many ways to abstract. Focusing on properties, phenol and isopropanol are both alcohols as they both contain a hydroxyl group. Focusing on behaviours, a pipette and pump are both liquid-transfer tools as both transport liquids. Focusing on representations, a checklist, flowchart or paragraph can describe how a new chemical or material was created, and an experimental method can be simplified using sensible labels (for example, 'products P1–10 were synthesized using method M1 and substrates S1–10'). For laboratory automation, the power of abstraction shines in cases where a workflow can be developed for a general case (for example, reagent source, transfer tool and reactor) rather than for each potential implementation (for example, syringe, syringe pump and packed bed reactor versus vial, pipette and heater shaker).

Despite the diversity of ways in which things can be abstracted[28], not all approaches are equally appropriate. When devices or operations are organized under the same abstraction, it is a statement that they are interchangeable in a specific context (for example, isopropanol and ethanol may be effectively interchangeable for disinfecting but not imbibing). Owing to the diversity of contexts in chemical and materials science, it is challenging to predict which details or distinctions matter. The context determines the nature of interactions; and in codification, these interactions are represented by a given 'interface' (Box 1). These interfaces define which interactions are permissible by defining a set of 'contracts'[29] (Box 1) that can be invoked and must be honoured.

Contracts

Within an experimental procedure, every step outlines a set of inputs and expected outputs, regardless of the level of abstraction present. Inputs can include material from previous steps in addition to process parameters (for example, reaction temperature and duration, the degree of mixing and so on). Outputs can include materials and observations or data as well as errors and side effects (for example, transfers will contaminate pipetting tips, a robotic arm may rearrange well plates when trafficking between instruments).

The creation of abstractions for experimental sciences requires the construction of new contracts for the abstracted form. For example, when abstracting a pipette, pump and hopper to honour a transfer contract, the contract could state that the source, destination and amount of material must be specified and that the result will be either an error or the specified amount of material being added to the destination. Note that this contract neglects the changes in the pipette or pump or hopper's state (Box 1; for example, becoming contaminated) as well as the changes in the source's state (for example, losing material).

It may seem logical, then, to fully specify every potentiality in the contract. However, as the number of entities that are involved in a contract increases, the more interdependent (coupled; Box 1) and, by extension, fragile the process may become. Coupling (Box 2), especially in complex systems, is observed to make the maintenance and analysis of code challenging and hinder the portability of code as actions become dependent on other entities that may or may not be present in another implementation[30,31]. For example, using flow rate to specify a reactor residence time requires knowing the relevant flow-path volumes to be translated to a reaction duration in a batch set-up. Similarly, trying to make a contract sufficiently general without involving other entities makes the contract less practical to use (consider the hyperbolically unhelpful contract 'something may happen').

As an illustrative example, the considerations for a contract between control software (contractee) and an HPLC controller (contractor) are discussed. Broadly, the contractee wishes to provide a sample in a vial or sample loop and to receive an analysis. Contractors and contractees can generally coordinate in three ways: providing information, making assurances and providing tools. For information, the contractee could provide the vial position, the HPLC method and

https://doi.org/10.1038/s44160-024-00649-8

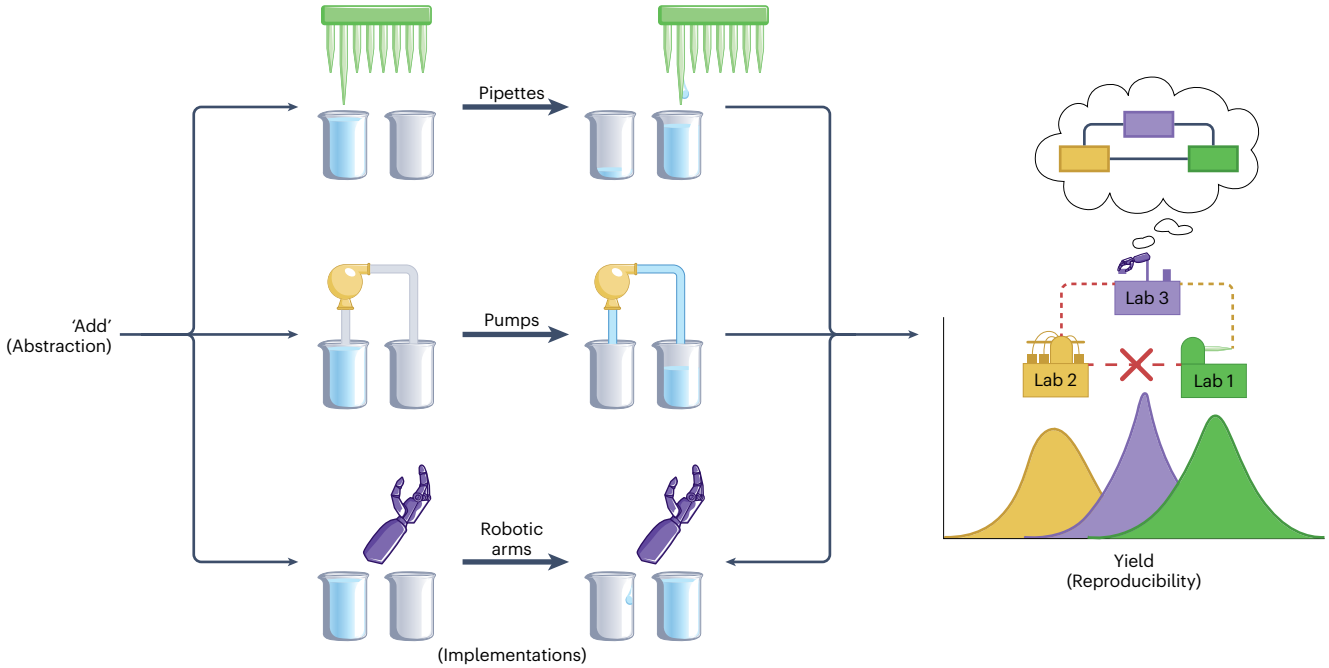


Fig. 2 | Various implementations of the same abstraction leading to different results. The broad abstraction 'Add', without any further parameterization or support can be realized in various ways. The fundamental qualities of these implementations can lead to challenges in reproducing experimental results—rendering these implementations as non-interchangeable. Without reproducible transferability, the dream of collaborative, automated research will not be possible.

where to save the data; the contractor could provide the results of its analysis and the status of the HPLC. For assurances, the contractee could guarantee that the sample will be in the vial or loop, that the method file exists and that the sample does not contain materials which may damage or clog the HPLC; similarly, the contractor could guarantee that, after each HPLC analysis, the column will be ready for another injection. For tools, the contractee could provide the sample valve controller (a subcontractor) so the contractor can properly time the injection. Concerning subcontractors, when the contractor itself makes a subcontract it becomes coupled: if the contractor took direct control of the sample valve, then this implementation could never work on a system without a sample valve. By providing a subcontractor as input, the contractee can fill in the contract with information it knows (or apply restrictions) and leave completing the contract up to the contractor. This makes the implementation of the contractor more general but increases the overhead—each invocation of the contract requires preparing and providing a subcontractor, and each subcontractor requires its own contract. Contracts must consider the contractee's goal and the contractor's requirements. The contractee could want a table of concentrations per compound, the unprocessed chromatogram or the selectivity and yield of a reaction. Given the goal, the contract must consider what is possible for the contractor and what information and tools the contractor requires (for example, to calculate the reaction yield, the contractee must provide the initial concentrations of reactants). The contract must specify how inputs and outputs are handled: should the contractor provide the data directly to the contractee or save it to a specific file or database; moreover, in what format (row- or column-major order, in what units and so on)? Finally, the contract must consider failures—which errors can be tolerated, are any of the assurances compromised by a failure?—and how these errors should be communicated between the contractor, contractee and subcontractors.

The best contracts are iteratively refined until all parties can complete their tasks by considering what information is known by each party, which tools are accessible to each party and what each party can guarantee going into and coming out of the contract. Contracts can be specific to an application (by only considering the answers to these questions on a single experimental platform) or general (by thinking about how others may have designed their experimental platforms). A contract specifying that yield and selectivity must be output to a file requires a computer where the control program possesses the security permissions for saving files and cannot be used by a different workflow that needs concentrations per component. Conversely, if the contract specifies the output to be a table of peak areas and retention times, additional contractors (and contracts) will be required to translate this table into what is needed by a different workflow.

Ultimately, contracts invite thinking about what the responsibilities of each component in a complex system are (and are not), what information or conditions are necessary to fulfil these responsibilities and how both successes and failures are to be detected and reported. Whereas contracts are a useful tool in the development of code, and by extension experimental procedures, they are not sufficient to guarantee reproducibility or interoperability.

Substitution

The ultimate purpose of abstractions which obey contracts is that different executors of a contract (for example, different experimental platforms) can be substituted without affecting the validity of the contract (for example, two collaborating laboratories observing the same material properties or a single laboratory migrating to a new vendor's hardware without needing to rewrite their experimental protocols). In a computer science context, any two equivalent algorithms that solve the same problem are interchangeable (despite differences in implementation and performance).

For experiments, the pipette, pump and hopper may appear interchangeable for material transfer; however, the quirks[32] of their physical implementation challenge the creation (or honouring) of contracts, and by extension challenge the guarantee of reproducibility upon substitution. As reported by Rauschen and colleagues[13], whereas the same quantity of water was specified as a quench on a Chemputer and a Kinova experimental system using χDL, the actual quantity dosed (and thus the product distribution) was different, as shown via NMR spectra;

https://doi.org/10.1038/s44160-024-00649-8

BOX 2

Coupling

Coupling can extend beyond software to physical experimental set-ups. A workstation where a central robotic arm transfers materials between modules, such as in the set-up reported by Gongora et al.[100], has each module mechanically coupled solely to the robotic arm (see image). As a result, if the arm malfunctions, the other modules may carry on (to an extent); similarly, if a single 3D printer malfunctions, the others may continue to make progress. Conversely, flow chemistry systems, although they provide improved reproducibility through enhanced process control when compared with batch chemistry, are highly coupled—for example, without flow-path isolation, if a single fluid-delivery pump fails, then all downstream processes and measurements are impacted. Beyond architecture, unit operations can exhibit coupling. Separations, chromatography in particular, are often highly coupled: chromatography will remove material from both a sample and from its solvent reservoirs, dilute and separate the sampled material and deposit new material in new locations—such as waste or a fraction collector. In addition, if the sample contains metals or is strongly basic it can damage the chromatography column, altering the efficacy of the chromatograph with future samples. Ultimately, the minimization of coupling is what enables truly flexible, modular experimental systems that are capable of diverse experiments[101].

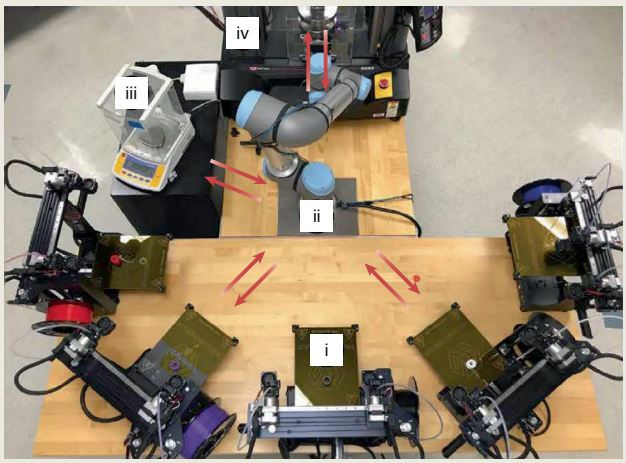


Image adapted from ref. 100 under a Creative Commons licence CC BY 4.0.

moreover, for a difficult, stochastic crystallization step, two Chemputer systems using different crystallization techniques produced different yields (87 and 47%) for a reaction intermediate.

In molecular discovery, where reactions are not optimized, a limited yield is a concern for completing multi-step syntheses and characterizations. If the well plate, batch platform reported by Koscher et al.[33] were translated to a pump-based system, such as a Chemputer[34], material losses due to dead volumes or purges may jeopardize the reproduction of a full experimental workflow. Either the entire process will need to be scaled up (incurring numerous experimental changes) or the researcher must accept running fewer or partial experiments. Conversely, a Chemputer-based protocol may need to be scaled down to run on a well plate system and adjusted for the inferior mixing of the well plates. Treating pipettes and pumps as interchangeable can result in problems with or discrepancies between the experimental workflows being hidden until execution—when resources have already been committed.

In a chemical synthesis context, consider the abstraction of 'solvent'. Both water and methanol are solvents; however, they are not substitutable in all reactions. Solvent effects can result in different activities, selectivities and yields in catalytic reactions. In the example provided by D'Agostino and co-workers[35], whereas the reaction will progress under water or methanol (interchangeable), they are not interchangeable in terms of yield and selectivity (even despite the two solvents being polar protic solvents, the same specific subtype of solvent). When specifying procedures executed by machines, care must be taken to use the correct level of abstraction (for example, does the reaction require a solvent or an acidic polar protic solvent?; by analogy, does the method require pumps or water-free, high-pressure pumps?).

Ideally, the side effects of using (overly) abstracted interfaces for different implementations could be resolved by making better implementations—for example, developing pumps with no dead volume, identifying chemically neutral and universally immiscible backing fluids for pipettes or developing a transfer technique that works with or without transfer solvents and can handle gases, liquids, solids, slurries, suspensions, foams and so on. In reality, however, such technologies do not, and may never, exist. More immediate improvements to repeatability may come from the development of more adaptive process control systems that leverage machine learning[36–38]. In addition, the repeatability of a chemical process can be improved by discovering more selective[39] or more fault-tolerant reactions[40]. Current laboratory automation technologies for (bio)chemistry and materials science can, and should, improve to address concerns of interoperability and reproducibility[41], but the language used to abstractly describe them must also shoulder some of that burden. To this end, improvements to automation will improve accuracy and repeatability, whereas improvements to the communication of experimental workflows will improve repeatability and reproducibility—and together will make automated experimentation more reliable. In addition, technologies and strategies for improving reproducibility which stem from outside the field of automation will still need to be communicated between (automated) laboratories. This communication (if written or digital) will require some level of abstraction to summarize material provenance[42], the procedure, the controller settings and expected behaviours.

Treating real devices as necessarily interchangeable because they are conceptually related can cause headaches as outcomes fail to be reproduced and problems are not identified until during execution of the experiments. Fundamentally, abstraction permits modifications to an experiment, and which changes are immaterial cannot be known a priori without a deep understanding of the underlying process: as the steps in a process become more abstracted, the guarantee of their reproducibility is diminished.

Exercise

Table 1 provides a prompt for the complexity of defining a sufficient but minimally coupled contract for the substitution of two different transfer tools. As an exercise, consider how a contract developed for the transfer of a material may change with the addition of (1) a hopper that operates on granulated or powdered solids and may not be able to pick up material (instead relying on a reservoir), (2) gravimetric (instead of volumetric) approaches that use partial-transfer or check-weight-differential cycles, (3) a robotic hand that can manipulate existing laboratory tools, such as beakers, scoopulas and gas bags, as would a human chemist or (4) the inclusion of slurries, which may interact negatively with tip diameters or pump heads.

Lessons learned from abstraction in computer science

The problem of granting interoperability through abstraction while incurring challenges to reproducibility is neither new nor solved. The

https://doi.org/10.1038/s44160-024-00649-8

Table 1 | Prompts for consideration in the design of a substitutable abstraction of material-transfer techniques

	Liquid-backed pipette	Pump
Valid source or destination[a]	Often deployed on a robot that can address source and destination independently.	Destinations and sources may be combinations of fixed or free, depending on the supporting equipment.
Valid transfer volume or mass	Tip and syringe sizes. Source or destination volumes.	Source or destination volumes.
Precision or accuracy of transfer	Tip and syringe sizes. Tip contact (above or submerged, touching a wall). Syringe motor precision and speed. Material viscosity, vapour pressure, surface tension and compressibility. Leading and trailing air gaps.	Tip contact (above or submerged, touching a wall). Motor precision. Material compressibility.
Amount of material taken from source	Same as deposited in destination.	At least as deposited in destination, depending on implementation.
Method of contamination	Residual material on pipettes, contact with backing fluid.	Mixing with pump fluid (if amount taken from source and amount deposited are equivalent).
Valid material phase	Liquid	Liquid
Example contract	**Input**: a source, destination, desired volume and routing subcontractor. **Actions**: use the subcontractor to locate the source or configure valves, remove material from the source, use the subcontractor to locate the destination or configure valves, deposit the specified quantity of material. **Output**: a destination with the specified volume of material. **Errors**: clogs, any error from the subcontractor.	
Notes	The volumetric units, the format of the locations and the interface for the subcontractor are not specified. The use of a subcontractor to configure valves or position pipettes is a good way of ensuring that the contract can be used by both flow and liquid handler set-ups. This contract fails to consider assurances such as there being sufficient material in the source or available space in the destination. In addition, there are no assurances about contamination—the actions could be modified to wash after every transfer to ensure no contamination. More complex cleaning (such as only cleaning when the transferred material changes) requires additional information about the material (not just its location), records of the history of transfers and information on how the material is transferred.	

[a]The dependence on supporting equipment implies, perhaps surprisingly, that the selection of sources and destinations should not be a responsibility of a transfer tool—in other words, a higher-order abstraction that can coordinate both material transfer and location selection is necessary.

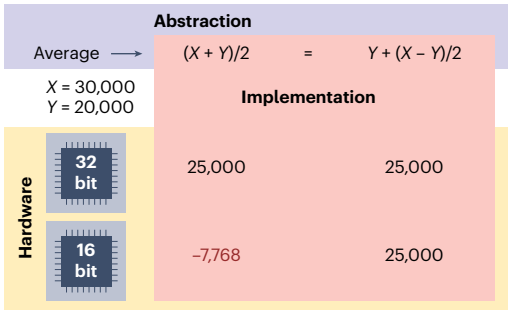


Fig. 3 | Disparity between abstractly equivalent but hardware-dependent implementations for a Java sorting algorithm. The sorting algorithm involves calculating the midpoint between a high index, X, and a low index, Y (both positive integers); the error arises because the intermediate term $(X + Y)$ can exceed the maximum value the hardware can store even when X, Y and their average are all valid numbers.

conditions for the interchangeability of objects (formally, behavioural subtyping[43]) were developed by Liskov and Wing three decades ago and are still under investigation[44,45]. Despite a mathematical (abstract) equivalence, an implementation-specific bug (Fig. 3) was found in Java sorting algorithms in 2006 (after avoiding detection for years)[46]. Enterprise systems[47], the Internet[48] (of things[49,50]) and robotics[51] face a similar problem where related elements cannot be abstracted to their core concept due to specific implementation details. Fortunately, experimentalists aiming to have codified experimental procedures can stand on the shoulders of these giants.

Below, we review some abstract programming language features which have been useful in computer science for addressing the problems that experimentalists face. These features centre around two ideas: (1) providing more operational detail than success or failure enables more nuanced control; and (2) experimental instructions (for example, 'heat at 90 °C and stir for 45 min') can be augmented by the results (for example, 'until solution turns clear'). Whereas the core lesson to be learned is that two implementations should not be grouped under one abstraction (for example, all material transfers into one standard 'transfer' instruction) unless the implementations are functionally identical in result, the more a language can support and communicate situational awareness (for example, sensor readouts, supplemental measurements or reports of observations) the better it can adjust a workflow to ensure the reproducibility of the results. A programming language for experiments should make the communicating intent, the technique and the expected results easy without involving clever (often inelegant or impenetrable) tricks.

Reactive approaches

Traditionally, experimental procedures are given imperatively (for example, do these steps in this order, and if something goes wrong, start over; Fig. 4) as this strategy suits the descriptive approach used in literature. However, with digital procedures, more nuanced, prescriptive workflows are possible. Rather than each operation reporting either success or an error, operations can be expected to report back with details of their operation—indicating complete or partial success and their observed changes of state[33,52,53]. As a result, the reporting of expected problems is incorporated into the contract without coupling to the manner by which the problems are addressed. A language that utilizes this approach of direct response needs to provide a way for responses to be captured and handled—the contract of abstracted operations must include sending a response. In existing automation languages (Fig. 5), the examples Fig. 5a–d,f are checklists without explicit syntax to handle errors—error handling is hidden within each step or by the coordinator going through the checklist, which prevents the user from implementing specialized recovery procedures. The example in Fig. 5g (LabVIEW) has 'wires' which show the transfer of results and error messages between instructions (non-graphical languages will typically use a 'result = instruction' syntax). Capturing responses is frequently achieved by having operations publish results to the caller (Box 1) or a communal resource (such as a database) (Fig. 4b) where some reactive logic can be implemented to course correct if necessary. In many ways, this approach is mirrored by laboratory technicians using a scale to measure 100 mg of reactant and then recording the actual amount (98.7 mg).

https://doi.org/10.1038/s44160-024-00649-8

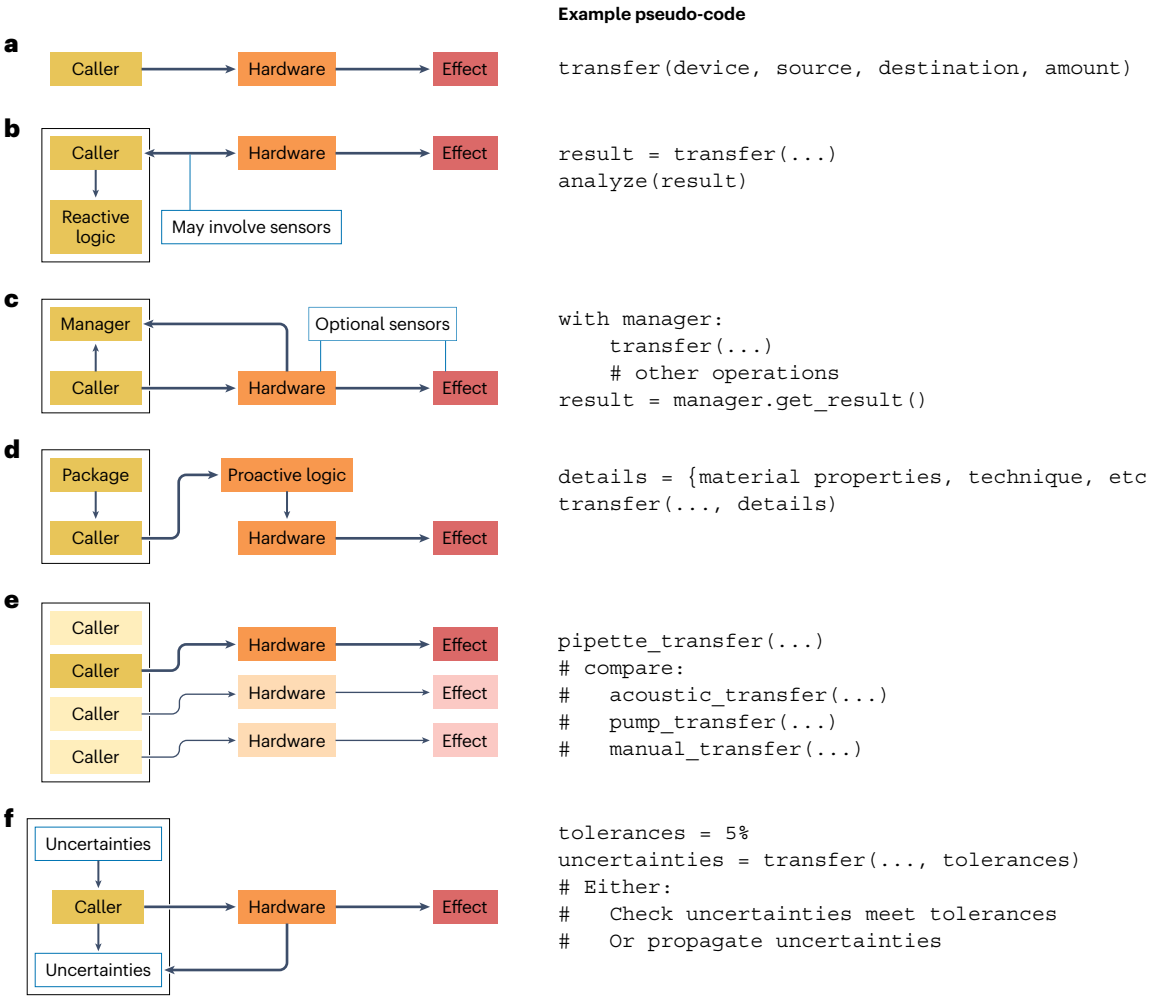


Fig. 4 | Visualizations of the approaches described. Generally, a caller (the part of the language that invokes some piece of hardware) sends a command to the hardware which, in turn, causes some effects (good, bad and neutral). An ellipsis is used for brevity on what information the callers may need. **a**, The status quo, an imperative approach where commands either succeed or fail. **b**, Commands report back with what happened. **c**, A supervised approach, where an external process can observe the reality of what happened. **d**, An interpretive approach, where specification details are processed in the context of the current hardware. **e**, A variant approach, where abstraction is limited to implementation assuming interchangeability between the hardware of that implementation (for example, all pipetting liquid handlers are interchangeable for pipetting transfers). **f**, A tolerance or uncertainty approach, where the limitations of the hardware and the tolerances of the workflow are processed along with the experiment. Throughout, lines preceded with a hash symbol (#) indicate comments.

In looking to design good contracts for abstracted operations, any sufficiently general contract will leave aspects of the experimental workflow underspecified; however, this slack can be picked up elsewhere. In many cases, a context manager[54,55] or error checker[56] (supervisor) is sufficient to address problems of a single operation changing the states of multiple entities. A supervisor is a separate process that takes notice of how an experiment's state may change between operations (such as checking the feasibility of transfers, recording the changes of material in a transfer or inserting proper washing and preparation steps on contamination), as shown in Fig. 4c. In addition, supervisors can manage metadata or environmental observations (ambient temperatures and humidity, for example). The use of context monitoring has seen success in languages such as Erlang[57], which adopts a 'let it crash' ideology for processes to defer error handling to higher-level processes that may be better suited for handling the error; similarly, Python[58] implements a 'with manager(details) as name:' syntax specifically to handle contractee assurances (such as checking for a file, opening it, monitoring for errors and closing the file). A single platform may also have multiple managers, enabling specialization in data-, hardware- and workflow-management[12,59]. The ORGANA system[60], for example, uses multiple computer vision processes to act as a supervisor for locating and positioning resources and for detecting anomalies to improve operational robustness. In the space between manual and automated experimentation, the rules and features of electronic laboratory notebooks[61] simulate some aspects of a supervisor but for human operators. Within extant languages for laboratory automation, those which use supervisors either provide global supervisors or hide single-step-specific supervisors within each operation. Future languages should consider providing syntax to handle more intermediate management (for example, the following steps must occur within a glovebox, using the same pipette tip or concurrently).

Although supervisors do not solve every problem, they do reduce the frequency (or impact) of errors during an experiment and may help detect errors in simulated operating modes. In this approach, the language needs a way to invoke and communicate with the supervisor. The monitoring or describing of operations provides aid in diagnosing problems and can enhance safeguards for the reproducibility of experimental results, although this may come at the cost of added complexity and inertia (one cannot proceed with just a hotplate for a pilot reaction: now a digital twin, database or sensor suite is needed). These techniques may synergize well with ontology- and event-based

https://doi.org/10.1038/s44160-024-00649-8

a ARChemist language

```
process:
  - state_name: stir_samples
    station:
      type: IkaPlateDigital
      id: 23
      operation:
        type: IKAStirringOp
        properties:
          stirring_speed: 1000
          duration: 300
    transitions:
      on_success: shake_samples
      on_fail: end_state
  - state_name: shake_samples
    station:
      type: ShakerPlateStation
      id: 22
      operation:
        type: ShakeOp
        properties:
          duration: 300
    transitions:
      on_success: analyse
      on_fail: end_state
```

b Koscher et al.[33]

```
'12':
  agent: robotic_arm
  operation: move_wellplate
  completed: 'no'
  container: hplc_plate
  details:
    target_destination: autosampler
  time_est: 30
  start_time: null
  end_time: null
'13':
  agent: liquid_chromatograph
  operation: run_analytical_batch
  completed: 'no'
  container: hplc_plate
  details:
    queue_number:
      - 3
      - 4
    update_plates:
      - filtrate_plate
      - hplc_plate
  time_est: 11200
  start_time: null
  end_time: null
```

c χDL

```
<Repeat repeats = "3">
  <Add
    vessel = "reactor"
    reagent = "acid"
    volume = "1 mL"
    stir = "True"
  />
  <Monitor
    target = "reactor"
    quantity = "pH"
    min = "5"
    max = "7"
  />
</Repeat>
<Add
  vessel = "reactor"
  reagent = "dichloromethane"
  volume = "30 mL"
  stir = "True"
/>
<Transfer
  from_vessel = "reactor"
  to_vessel = "separator"
  volume = "all"
/>
```

d Autoprotocol (via Python)

```
LH = LiquidHandle
LH.builders.transport(
  volume=Unit(1, "uL"),
  density=None,
  pump_override_volume=Unit(2, "uL"),
  flowrate=LH.builders.flowrate(
    target=Unit(10, "uL/s")
  ),
  delay_time=Unit(0.5, "s"),
  mode_params=LH.builders.mode_params(
    liquid_class="air",
    position_z=LH.builders.position_z(
      reference="preceding_position"
    )
  )
)
```

e MAOS language

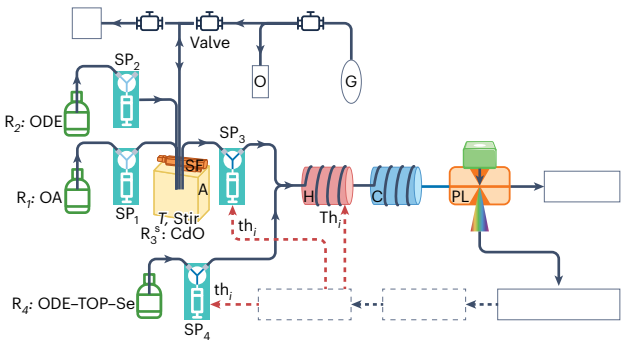
MAOS code

$$PL \mid (Pc \bullet SP_3 + R_4 \bullet SP_4) \times H \times C$$

where $Pc =$

$$[R_3^s \bullet SF \times A + (R_1 \bullet SP_1 + R_2 \bullet SP_2) \times (A \cup V) \times (A \cup G \cup O)]$$

Flow diagram

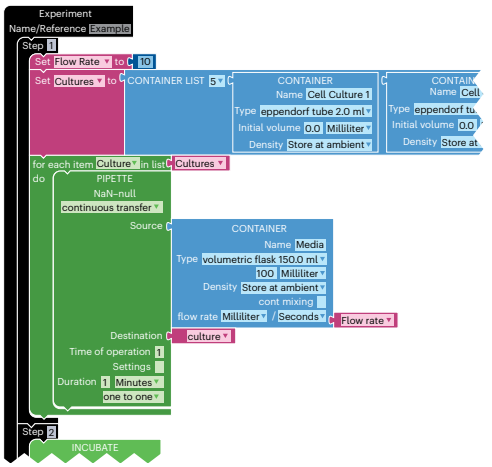


Legend

PL	(assay)	Photoluminescence
Pc	(variable)	Reactor effluent
$x \bullet y$	(operation)	Transfer reagent x via y
$x + y$	(operation)	Combine reagents x and y
$x \times y$	(operation)	React x in environment y
$x \mid y$	(operation)	Characterize y via x
$x \cup y$	(operation)	Combine environments x and y

f BioBlocks

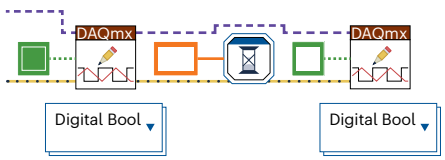
BioBlocks code



Schematic

g LabVIEW

Code



Schematic

Fig. 5 | Example excerpts of different languages for prescribing automated experiments. Excerpts focus on the operations of the workflow and omit the declaration of materials and resources in their respective workflows for brevity. **a**, The language used on the ARChemist platform to describe the stirring and shaking of samples. The use of 'on_success' and 'on_fail', which point to which step is performed next, enable branching workflows, although in this version, however, there are no protections against infinite loops. Code from refs. 91,92. **b**, The language used by Koscher et al. to describe the transfer of a well plate into a chromatograph and subsequent analytical separation. Each step may use conditional logic and loops by modifying the workflow via Python scripts, increasing the language's expressive power but making the workflow opaque. Code from ref. 33. **c**, An excerpt of χDL to describe the neutralization of a sample before transfer to a separator. Loops are given a well-defined structure that permits exits on the experimental conditions and attempt limit. Code from ref. 93. **d**, Autoprotocol (in Python) to describe the liquid-transfer parameters for a liquid handler. The source of the material in the transfer step is inferred from the 'preceding_location' specification. Code from ref. 94. **e**, The MAOS language to describe an entire flow chemistry system with mathematical-like abstraction[7] (the flow system is provided as a reference). PL, photoluminescence; Pc, reactor effluent; SP, syringe pump; R, reagent; R^s, solid reagent; H, heating module; C, cooling module; SF, solid feeding module; A, reactor module; V, vacuum pump; G, gas supplier; O, gas treatment; ODE, 1-octadecene; OA, oleic acid; *T*, temperature; CdO, cadmium oxide; Th, heating temperature; th, heating time; TOP, trioctylphosphine; Se, selenium. **f**, A visual BioBlocks description of administering media to multiple cell cultures[95]. Aspects of the interfaces between operations are visualized by the puzzle-piece connectors between statements and the selection boxes within statements. **g**, Example LabVIEW[8] code that describes a half-second relay pulse to trigger the start of a chromatographic analysis. The dashed purple line contains information about which relay to use; the dotted yellow line carries error messages. This language enables a more flowchart description of operations (as opposed to the more linear representations of the previous examples) and represents hardware and software interfaces with connection ports on each block, which the user connects with 'wires'. DAQ, National Instruments Data Acquisition system; DAQmx, a DAQ API for LabVIEW; T, the boolean value 'True'; F, the boolean value 'False'. Comparisons between these, and additional, languages for the same operation are provided in Supplementary Section 2. Panel **e** adapted from ref. 7 under a Creative Commons licence CC BY 4.0.

approaches to laboratory automation data management[42,62–64], through their alignment with the publishing of results to supervising processes.

Interpretive approaches

In contrast to the aforementioned, reactive, techniques, a more proactive approach can be taken. It is possible to provide each operation with as much information (but not necessarily instruction) as possible (Fig. 4d) at the onset of a task. This information can be sent explicitly at the invocation of the operation or implicitly by providing a link to a resource, such as a database or sensor, instead. The executor of the operation can filter (if the command is overspecified) or enrich (provided with sufficient information) these data as needed[47] and adapt its approach (strategize[65]) to meet operational objectives. This strategizing can include choosing between different implementations (for example, performing a separation via filtration or liquid–liquid extraction). The explicit approach, where supplemental information is sent directly, can be simplified using conveniently formatted data packages[47]; the implicit approach can utilize lightweight messages and keep the writing of instructions simple[47] but requires the consolidation of details elsewhere such as a database.

The automation control architectures of Koscher et al.[33] (Fig. 5b) and Statt et al.[64] demonstrate convergent evolution towards the same design patterns (Box 1), where devices communicate to a central server instead of each other. The two differ in how messages are transferred: the former has a blend of forwarding messages and has systems check for updates, whereas the latter has all members checking for notifications. Such centralization of information is seen in other automation architectures[2,66] as it enables a simplification of intersystem communication and better isolates modules (reduces coupling).

Laboratory automation languages such as Autoprotocol (Fig. 5d), AnIML and PyLabRobot, for example, provide highly detailed operational parameters with their commands. This information can then be used to construct a functionally identical operation (aiding in reproducibility). Unactionable information may optionally be ignored, hindering reproducibility but extending interoperability—for example, ignoring a pipette aspiration speed parameter because the hardware does not provide that level of control. Leveraging the potential for graphical languages[8,11] (Fig. 5f,g), a highly detailed, hierarchical structure such as that of Community Resource for Innovation in Polymer Technology (or CRIPT)[67] could, with modification, enable workflows to be abstracted or replicated as needed by having implementations only explore details to a desired depth.

Highly descriptive laboratory automation languages need a standardized way to contain and interpret these instructions as well as a way to understand the capabilities of each executor. Towards the aim of implementation-agnostic reproducibility in (bio)chemical and materials sciences, the descriptions could be shifted in focus from the implementation of actions (that is, 'imperative programming' (Box 1); for example, Autoprotocol's 'provision' specifies a quantity of material to be added to a well[2]) to the results of those actions (that is, 'declarative programming' (Box 1); for example, the platform of Koscher et al. specifies the target concentration profiles of a well and has the liquid handler's controller solve for what quantities of material need to be added[33]). Between versions 0.5 and 2.0, χDL implemented the idea of 'do until' (monitor) as a half-way point between these approaches: for example, adding acid 1 ml at a time until a specified pH is reached[68] (Fig. 5c). A declarative paradigm would, in theory, be excellent at ensuring reproducible results and could take inspiration from existing declarative languages (for example, HTML, SQL, Lisp, Prolog and mathematical notation); in practice, however, this would require the creation of some program that could solve how to achieve those results with the given hardware (and the problem of ensuring that reproducibility and transferability loops back on itself). There is potential for first-generation declarative paradigms in flow systems, whereby simple processes can be specified in terms of dimensionless parameters and models for which mathematical formulae exist to perform the solving step—or determine a priori if an experiment is even possible on the system.

Limited abstraction

A language could compromise out-of-the-box interoperability for robust and safe abstraction by restricting abstractions only as far as they are interchangeable. In this approach, rather than a single abstracted 'transfer' operation, there may be multiple (Figs. 4e and 5a). In this model, any changes made to the experimental workflow so that it can be executed on different hardware must be made consciously. When these changes are recorded, it enables a scientist-programmer to properly characterize experimental reproducibility with respect to the changes made to the workflow.

Unfortunately, this approach is most similar to the status quo and suffers from a known major setback: curation. To compensate for the reduction in interoperability, a community-accessible library comprising each version of an operation must be created, validated and maintained. Moreover, the number of versions must be kept manageable and each version must have good and proper documentation for each version to be found and used correctly.

Should the curation problem be addressed, however, this minimally abstracted approach holds three notable benefits. First, it enables problems in a workflow to be detected before execution (for example, receiving a warning for missing or incompatible hardware). Second, as there are fewer layers of abstraction, errors during execution are more concretely tied to the operation on which they occur—facilitating troubleshooting. Third, prototype methods can be developed and used freely—reducing the friction to starting up an automated laboratory or continually evolving new capabilities.

Emulating unreliability

From a hardware perspective, it can be possible to catalogue the capabilities (and lack thereof) of each implementation. Similar to cloud-computing strategies[69], these labels can be taken to develop a lowest-common-denominator implementation such that diverse hardware will all produce the same (if suboptimal) results. Crucially, this approach should provide the ability to turn this emulation on (when reproducibility is the focus) or off (when exploration or optimization is the focus). For example, while dual-syringe pumps from different manufacturers allow each syringe to operate at different flow rates and directions and to start and stop independently, the dual-syringe pumps from one manufacturer cannot configure a syringe while the other is running. As such, any collaboration between two research laboratories that use these syringe pumps would need to follow the order of operations set by the hardware that does not allow simultaneous configuration.

Alternatively, these labels of capability can be taken to record the experimental precision/tolerance and bias of each implementation[70]. This approach embeds repeatability information into the procedure, which in turn can aid in enhancing reproducibility (or at least quantifiably explaining failures of reproducibility) (Fig. 4f). Not many programming languages today use this strategy, as the reliability of processors has greatly increased since the days of vacuum tubes and as unreliabilities in network hardware could be tolerated or circumvented with best-practice standards or heuristics and powerful support systems beyond the programming language itself—for example, compilers or interpreters (Box 1), integrated developer environments[7,9,71] and emulators. However, with the stochasticity inherent in chemical and material processes and the variable precision and accuracy of commercial automation hardware, this specification of code reliability becomes increasingly relevant in laboratory automation.

A proactive application of this approach is hindered by the unfortunate reality that the precision and accuracy of automated systems are not necessarily known—a liquid handler may be very precise with

aqueous solutions but not with low-surface-tension (for example, hexanes) or volatile (for example, chloroform) solvents. To proactively use uncertainties and propagate estimated errors, automated systems will first need to be rigorously characterized for each application. Whereas it is reasonable to expect a laboratory to characterize the pipetting accuracy of its liquid handler with one or two relevant solvents, for platforms that use complex mixtures or handle the products of other transformations and treatments, the relevant space rapidly becomes too large to fully characterize. This problem would imply that a retroactive approach is more appropriate: such as pulling random samples to validate that the system is operating within tolerances. To this end, uncertainty-oriented languages could be combined with reactive or supervised approaches to better qualify experimental workflows. As previously mentioned, instructions may specify one mass (100 mg) but the actual mass (98.7 mg) may be different. In this case, predicting uncertainty is unnecessary as the actual error can be measured instead. Strict encoding of tolerances would enable a robust determination of whether this −1.3% error is acceptable. This issue does raise the question of whether someone seeking to replicate this study should target 100 mg or 98.7 mg—and how dependent quantities should be adjusted. Languages that focus on unreliability may need to shift from representing quantities as a singular number (100 mg) to a more general object that encodes the specified and realized values of each parameter.

Outlook

The role of programming languages that can standardize the communication of experimental workflows for automated platforms is to provide both transferability and reproducibility of experimental results. With reproducible, digitalized experiments, efforts towards collaborative research (such as networks of self-driving laboratories[72]) can be realized—accelerating research and cultivating trust in automated laboratory technologies. Whereas the guarantee of reproducibility should be a major goal of any language for codifying experimental procedures in the (bio)chemical and materials sciences, it is not the only goal that must be considered. A language must also be accessible and efficient. Non-computer scientists seeking to codify an experimental procedure need to be able to read, write and understand a language without spending years studying. There are undoubtedly more lessons that could be extracted from computer science for automated experimentation. The lessons presented here are not mutually exclusive, and multiple strategies can be combined to meet the challenges of the community.

In the context of larger studies using codified experimental workflows, such as cloud, self-driving, autonomous or high-throughput laboratories, the nature of what is needed of a programming language can change[33,73–75]. Autonomous exploration, automated optimization and cross-validation workflows each implicate different allowable changes to an experimental workflow. Autonomous systems need freedom (ambiguity) to design and optimize their approach to realize (or abandon) goals[15]; conversely, when the objective is to leverage existing work to save time or to cross-validate another laboratory's work, concrete specifics are needed instead. A singular, unified language would need to simultaneously describe goals and implementations, instructions and expectations, and uncertainties and tolerances. Much as how programming languages have continued to be invented and developed, it is likely that multiple languages will form the basis of the laboratory automation ecosystem. It would be beneficial, given the development and diversity of laboratory automation languages and architectures, to survey the academic community and industry partners at this nascent stage as to why each group elected to use existing work or create something new, and what alternatives were considered and why they were ultimately adopted or passed up. With such information, the evolution of the laboratory automation ecosystem can be guided towards a minimal set of maximally useful languages.

Software can leverage a small, standardized number of highly reliable central-processing-unit-level commands to create general translators between programming languages and machine instructions. Laboratory automation, however, has a plurality of vendor interfaces (and, more importantly, capabilities), which require users to first build their own controllers for everything that is being automated[32], and which fundamentally challenge the ability to create contracts that can be honoured regardless of the hardware vendor.

Standardizing the capabilities of commercial hardware and software is not feasible—the improvement, addition and specialization of features is crucial for these businesses to survive. As a result, workflows that rely on established (common) techniques may be able to change vendors more freely than workflows that leverage cutting-edge or specialized technology to achieve mission-specific goals. Consequentially, much attention has shifted to the standardization of interfaces with commercial laboratory automation hardware and software (also known as the API).

At present, laboratory automation solutions have internally consistent interfaces[12,76–78] (which often leverage existing interfacing tools such as FastAPI[77,79], SiLA (Standardization in Lab Automation)[80] or ROS (Robot Operating System)[59,81]) but require users to build the connective tissue between the vendor's API and the controller's interface—often requiring supplemental code to ensure that the user-made code can fulfil the contracts requested by the controller. Laboratory automation vendors may struggle to abide by a universal standard—given the diversity of users, namely, the diversity of contexts (which extend beyond academic research), a solution for one group may create a problem for another[82]. Moreover, changes to interfaces and data formats requires the rewriting of existing contracts—a massive effort and risk for a business. Even if all vendors used the same interface, many users would still need to write code to connect these interfaces to their preferred language (such as Python, LabVIEW or a bespoke language). Whereas consistent interfaces or APIs would represent a considerable quality-of-life improvement, standardized APIs are ultimately secondary to documentation. Proper documentation for an instrument, API and data formats is what formally describes both how to interact (interface information) and the underlying capabilities (contractual information). By providing better documentation and exposing more of the 'under the hood' features of the equipment, it would make it easier for individuals to write and share case-specific solutions. On the pain point of impenetrable data formats, if commercial software cannot provide data in an open-source format, the documentation should explain how to read the proprietary format. To this end, continued collaboration with industry partners to produce official and accurate documentation, which covers all of a system's underlying functionality, and to use less opaque data-transfer protocols will help to advance laboratory research automation.

As a final example, abstractions that aid in modularization (minimal coupling and true interchangeability) facilitate the testing of unit operations. When each process can be isolated, it enables core functionality and the specific connections between processes to be thoroughly examined. In the laboratory setting, this can permit the 'a la carte' validation of different reproducibility metrics—for example, does the observed product distribution of a given reactor depend on the subsequent separation technique?, how do the accuracy and precision of a flow system's pumps depend on fluid properties?, what is the quality of mixing or dispersion in and between modules? and so on. Taken as a whole, the community can inspect the dominant abstractions in (bio) chemical and materials sciences to develop general test suites that can help to catch problems early during development, help to communicate expectations of repeatability and reproducibly to the public[73,83,84], inform and guide the development of improvements to laboratory automation and, for self-driving laboratories, act as a driver's test[85].

Towards characterizing reproducibility, there are two general approaches. Laboratories could collaborate such that they cross-validate their workflows before publication. Alternatively, automated workflows could be required to report estimations for

reproducibility before publication—similarly to how a chemist reporting a novel compound must provide mass spectrometry and NMR spectroscopy data in most journals. The former approach has the benefit of fostering collaborations and bringing in more diverse ideas for improving automated experimental platforms. This approach, however, may be difficult to properly fund, given current grant structures, and risks over-centralizing power to a handful of established laboratories. The latter approach, the empirical estimations of reproducibility, could include performing tests where components of an automated set-up are replaced—such as changing the backing fluid of a liquid handler; using different pipette sizes, tube diameters or well plate geometries; or changing chemical vendors or intentionally spiking reagents with impurities. By analysing the overlap in repeatability test results between the two configurations, the reproducibility between similar platforms can be estimated. This approach is restricted to affordable changes—it would be prohibitively expensive to purchase duplicates of substantial hardware, such as syringe pumps or liquid handlers, especially for laboratories attempting to enter into the space of experimental automation.

The reproducibility of results must be addressed in the automated laboratory sciences. Before larger questions of democratization or federalization of self-driving experimentation can be approached[83,86,87], the basic unit operations of experiments need robust digital representations. Given the diversity of hardware capabilities, the creation of laboratory languages (and the computer processes that translate them into machine instructions) may require that their interpreters are extensibly configured for each laboratory—such as software requiring hardware-specific compilation. Even if fundamental differences in control may necessitate different languages for batch and flow chemistry or for single- versus federated-laboratory workflows, once reproducibility can be addressed in each domain, then it may be possible to develop 'transpilers'—devices that translate code between languages. In this regard, the success of machine learning for programming language translation[5,12,88–90] presents a tantalizing opportunity for learning the mappings between batch and flow chemistry, solid- and solution-phase processes or local and decentralized orchestration.

Given the precedent of multiple languages in computer science and language translators, this may represent the most reasonable future for laboratory automation. The design of a universal programming language would probably be too complicated (or too cumbersome) for the average user. Instead, by relying on translation, an experimentalist can express their ideas in a manner that is familiar to them but still share these ideas with others—thus inviting diverse perspectives on matters of experimentation.

References

1. *Autoprotocol* (Strateos Inc., 2021); https://autoprotocol.org/
2. Pendleton, I. M. et al. Experiment Specification, Capture and Laboratory Automation Technology (ESCALATE): a software pipeline for automated chemical experimentation and data management. *MRS Commun.* **9**, 846–859 (2019).
3. Leonov, A. I. et al. An integrated self-optimizing programmable chemical synthesis and reaction engine. *Nat. Commun.* **15**, 1240 (2024).
4. Documentation center. *Emerald Cloud Lab* https://www.emeraldcloudlab.com/documentation/ (Emerald Cloud Lab, Inc, 2023).
5. Boiko, D. A., MacKnight, R., Kline, B. & Gomes, G. Autonomous chemical research with large language models. *Nature* **624**, 570–578 (2023).
6. Schäfer, B. A., Poetz, D. & Kramer, G. W. Documenting laboratory workflows using the Analytical Information Markup Language. *JALA* **9**, 375–381 (2004).
7. Li, J., Tu, Y., Liu, R., Lu, Y. & Zhu, X. Toward "on-demand" materials synthesis and scientific discovery through intelligent robots. *Adv. Sci.* **7**, 1901957 (2020).
8. LabVIEW. *LabVIEW Wiki* (National Instruments, 2024) https://labviewwiki.org/wiki/LabVIEW
9. Bartley, B. et al. Building an open representation for biological protocols. *ACM J. Emerg. Technol. Comput. Syst.* **19**, 1–21 (2023).
10. Ananthanarayanan, V. & Thies, W. Biocoder: a programming language for standardizing and automating biology protocols. *J. Biol. Eng.* **4**, 13 (2010).
11. Gupta, V., Irimia, J., Pau, I. & Rodríguez-Patón, A. BioBlocks: programming protocols in biology made easier. *ACS Synth. Biol.* **6**, 1230–1232 (2017).
12. Wierenga, R. P., Golas, S. M., Ho, W., Coley, C. W. & Esvelt, K. M. PyLabRobot: an open-source, hardware-agnostic interface for liquid-handling robots and accessories. *Device* **1**, 100111 (2023).
13. Rauschen, R., Guy, M., Hein, J. E. & Cronin, L. Universal chemical programming language for robotic synthesis repeatability. *Nat. Synth.* **3**, 488–496 (2024).
14. Leong, C. J. et al. An object-oriented framework to enable workflow evolution across materials acceleration platforms. *Matter* **5**, 3124–3134 (2022).
15. Canty, R. B., Koscher, B. A., McDonald, M. A. & Jensen, K. F. Integrating autonomy into automated research platforms. *Digit. Discov.* **2**, 1259–1268 (2023).
16. Alexandron, G., Armoni, M., Gordon, M. & Harel, D. Scenario-based programming: reducing the cognitive load, fostering abstract thinking. In *Companion Proc. 36th International Conference on Software Engineering* 311–320 (Association for Computing Machinery, 2014); https://doi.org/10.1145/2591062.2591167
17. Fidler, F. & Wilcox, J. in *The Stanford Encyclopedia of Philosophy* (ed. Zalta, E. N.) (Metaphysics Research Lab, Stanford University, 2021).
18. Mandel, J. Repeatability and reproducibility. *J. Qual. Technol.* **4**, 74–85 (1972).
19. Feitelson, D. G. From repeatability to reproducibility and corroboration. *ACM SIGOPS Oper. Syst. Rev.* **49**, 3–11 (2015).
20. Pijper, B. et al. Addressing reproducibility challenges in high-throughput photochemistry. *JACS Au* https://doi.org/10.1021/jacsau.4c00312 (2024).
21. Epps, R. W. et al. Artificial chemist: an autonomous quantum dot synthesis bot. *Adv. Mater.* **32**, 2001626 (2020).
22. Bateni, F. et al. Smart Dope: a self-driving fluidic lab for accelerated development of doped perovskite quantum dots. *Adv. Energy Mater.* **14**, 2302303 (2024).
23. Leeman, J. et al. Challenges in high-throughput inorganic materials prediction and autonomous synthesis. *PRX Energy* **3**, 011002 (2024).
24. Sayre, F. & Riegelman, A. The reproducibility crisis and academic libraries. *Coll. Res. Libr.* https://doi.org/10.5860/crl.79.1.2 (2018).
25. Leins, D. A., Haase, S. B., Eslami, M., Schrier, J. & Freeman, J. T. Collaborative methods to enhance reproducibility and accelerate discovery. *Digit. Discov.* **2**, 12–27 (2023).
26. Liskov, B. & Zilles, S. Programming with abstract data types. *ACM SIGPLAN Not.* **9**, 50–59 (1974).
27. Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM* **15**, 1053–1058 (1972).
28. Parnas, D. L., Shore, J. E. & Weiss, D. Abstract types defined as classes of variables. *ACM SIGPLAN Not.* **11**, 149–154 (1976).
29. Meyer, B. Applying 'design by contract'. *Computer* **25**, 40–51 (1992).
30. Stevens, W. P., Myers, G. J. & Constantine, L. L. Structured design. *IBM Syst. J.* **13**, 115–139 (1974).
31. Taube-Schock, C., Walker, R. J. & Witten, I. H. Can we avoid high coupling? In *ECOOP 2011 – Object-Oriented Programming* (ed. Mezini, M.) 204–228 (Springer, 2011); https://doi.org/10.1007/978-3-642-22655-7_10

32. Christensen, M. et al. Automation isn't automatic. *Chem. Sci.* **12**, 15473–15490 (2021).
33. Koscher, B. A. et al. Autonomous, multiproperty-driven molecular discovery: from predictions to measurements and back. *Science* **382**, eadi1407 (2023).
34. Karafiludis, S., Ryll, T. W., Buzanich, A. G., Emmerling, F. & Stawski, T. M. Phase stability studies on transition metal phosphates aided by an automated synthesis. *CrystEngComm* **25**, 4333–4344 (2023).
35. D'Agostino, C. et al. Understanding the solvent effect on the catalytic oxidation of 1,4-butanediol in methanol over Au/TiO_2 catalyst: NMR diffusion and relaxation studies. *Chem. Eur. J.* **18**, 14426–14433 (2012).
36. Pomberger, A. et al. Automated pH adjustment driven by robotic workflows and active machine learning. *Chem. Eng. J.* **451**, 139099 (2023).
37. Nian, R., Liu, J. & Huang, B. A review on reinforcement learning: introduction and applications in industrial process control. *Comput. Chem. Eng.* **139**, 106886 (2020).
38. Maffettone, P. M. et al. Gaming the beamlines—employing reinforcement learning to maximize scientific outcomes at large-scale user facilities. *Mach. Learn. Sci. Technol.* **2**, 025025 (2021).
39. Martens, J. A., Perez-Pariente, J., Sastre, E., Corma, A. & Jacobs, P. A. Isomerization and disproportionation of *m*-xylene: selectivities induced by the void structure of the zeolite framework. *Appl. Catal.* **45**, 85–101 (1988).
40. Molyneux, S. & Goss, R. J. M. Fully aqueous and air-compatible cross-coupling of primary alkyl halides with aryl boronic species: a possible and facile method. *ACS Catal.* **13**, 6365–6374 (2023).
41. Jessop-Fabre, M. M. & Sonnenschein, N. Improving reproducibility in synthetic biology. *Front. Bioeng. Biotechnol.* **7**, 18 (2019).
42. Bai, J. et al. From platform to knowledge graph: evolution of laboratory automation. *JACS Au* **2**, 292–309 (2022).
43. Liskov, B. H. & Wing, J. M. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.* **16**, 1811–1841 (1994).
44. Hähnle, R., Kamburjan, E. & Scaletta, M. in *Active Object Languages: Current Research Trends* (eds de Boer, F. et al.) 289–322 (Springer, 2024); https://doi.org/10.1007/978-3-031-51060-1_11
45. Giordano, G. et al. On the adoption and effects of source code reuse on defect proneness and maintenance effort. *Empir. Softw. Eng.* **29**, 20 (2023).
46. Bloch, J. Extra, extra – read all about it: nearly all binary searches and mergesorts are broken. *Google Research* https://blog.research.google/2006/06/extra-extra-read-all-about-it-nearly.html (2006).
47. Hohpe, G. & Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* (Addison-Wesley, 2003).
48. Melloul, L. & Fox, A. Reusable functional composition patterns for Web services. In *Proc. IEEE International Conference on Web Services* (IEEE, 2004); https://ieeexplore.ieee.org/abstract/document/1314775
49. Tkaczyk, R. et al. Cataloging design patterns for internet of things artifact integration. In *2018 IEEE International Conference on Communications Workshops (ICC Workshops)* 1–6 (IEEE, 2018); https://doi.org/10.1109/ICCW.2018.8403758
50. Ramadas, A., Domingues, G., Dias, J. P., Aguiar, A. & Ferreira, H. S. Patterns for things that fail. In *Proc. 24th Conference on Pattern Languages of Programs* 1–10 (The Hillside Group, 2017).
51. Nesnas, I. A. D. in *Software Engineering for Experimental Robotics* (ed. Brugali, D.) 31–70 (Springer, 2007); https://doi.org/10.1007/978-3-540-68951-5_3
52. Rees-Hill, J. A. *Error Handling Approaches in Programming Languages* (Oberlin College, 2022).
53. Erwig, M. & Ren, D. Monadification of functional programs. *Sci. Comput. Program.* **52**, 101–129 (2004).
54. Salvaneschi, G., Ghezzi, C. & Pradella, M. Context-oriented programming: a software engineering perspective. *J. Syst. Softw.* **85**, 1801–1817 (2012).
55. Cardozo, N. & Mens, K. Programming language implementations for context-oriented self-adaptive systems. *Inf. Softw. Technol.* **143**, 106789 (2022).
56. Carbin, M., Misailovic, S. & Rinard, M. C. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proc. 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* 33–52 (Association for Computing Machinery, 2013); https://doi.org/10.1145/2509136.2509546
57. Armstrong, J. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology, Stockholm (2003).
58. *The Python Language Reference* (Python Software Foundation, 2024); http://python.org
59. Fakhruldeen, H., Pizzuto, G., Glowacki, J. & Cooper, A. I. ARChemist: Autonomous Robotic Chemistry system architecture. In *Proc. 2022 International Conference on Robotics and Automation (ICRA)* 6013–6019 (IEEE, 2022); https://doi.org/10.1109/ICRA46639.2022.9811996
60. Darvish, K. et al. ORGANA: a robotic assistant for automated chemistry experimentation and characterization. Preprint at https://arxiv.org/abs/2401.06949 (2024).
61. Higgins, S. G., Nogiwa-Valdez, A. A. & Stevens, M. M. Considerations for implementing electronic laboratory notebooks in an academic research environment. *Nat. Protoc.* **17**, 179–189 (2022).
62. Statt, M. J. et al. ESAMP: event-sourced architecture for materials provenance management and application to accelerated materials discovery. *Digit. Discov.* **2**, 1078–1088 (2023).
63. Duke, R., McCoy, R., Risko, C. & Bursten, J. R. S. Promises and perils of big data: philosophical constraints on chemical ontologies. *J. Am. Chem. Soc.* https://doi.org/10.1021/jacs.3c11399 (2024).
64. Statt, M. J., Rohr, B. A., Guevarra, D., Suram, S. K. & Gregoire, J. M. Event-driven data management with cloud computing for extensible materials acceleration platforms. *Digit. Discov.* **3**, 238–242 (2024).
65. Jung, E., Cho, I. & Kang, S. M. An agent modeling for overcoming the heterogeneity in the IoT with design patterns. In *Mobile, Ubiquitous, and Intelligent Computing: MUSIC 2013* (eds Park, J. J. et al.) 69–74 (Springer, 2014); https://doi.org/10.1007/978-3-642-40675-1_11
66. Green, D. V. S. et al. BRADSHAW: a system for automated molecular design. *J. Comput. Aided Mol. Des.* **34**, 747–765 (2020).
67. Walsh, D. J. et al. Community Resource for Innovation in Polymer Technology (CRIPT): a scalable polymer material data structure. *ACS Cent. Sci.* **9**, 330–338 (2023).
68. *XDL Documentation* (Cronin Group, University of Glasgow, 2022); https://croningroup.gitlab.io/chemputer/xdl/
69. John, W. et al. The future of cloud computing: highly distributed with heterogeneous hardware. *Ericsson Technology Review* (12 May 2020).
70. Carbin, M. & Misailovic, S. in *Foundations of Probabilistic Programming* (eds Silva, A. et al.) 533–568 (Cambridge Univ. Press, 2020); https://doi.org/10.1017/9781108770750.016
71. Craven, M., Keenan, G., Khan, A., Lee M. & Wilbraham L. *ChemIDE* (Cronin Group, University of Glasgow, 2021); https://croningroup.gitlab.io/chemputer/xdlapp/

72. Delgado-Licona, F. & Abolhasani, M. Research acceleration in self-driving labs: technological roadmap toward accelerated materials and molecular discovery. *Adv. Intell. Syst.* **5**, 2200331 (2023).
73. Bennett, J. A. et al. Autonomous reaction Pareto-front mapping with a self-driving catalysis laboratory. *Nat. Chem. Eng.* **1**, 240–250 (2024).
74. Fitzpatrick, D. E., Battilocchio, C. & Ley, S. V. A novel internet-based reaction monitoring, control and autonomous self-optimization platform for chemical synthesis. *Org. Process Res. Dev.* **20**, 386–394 (2016).
75. Li, J. et al. Autonomous discovery of optically active chiral inorganic perovskite nanocrystals through an intelligent cloud lab. *Nat. Commun.* **11**, 2046 (2020).
76. Gromski, P. S., Granda, J. M. & Cronin, L. Universal chemical synthesis and discovery with 'the Chemputer'. *Trends Chem.* **2**, 4–12 (2020).
77. Rahmanian, F. et al. Enabling modular autonomous feedback-loops in materials science through hierarchical experimental laboratory automation and orchestration. *Adv. Mater. Interfaces* **9**, 2101987 (2022).
78. Sim, M. et al. ChemOS 2.0: an orchestration architecture for chemical self-driving laboratories. *Matter* https://doi.org/10.1016/j.matt.2024.04.022 (2024).
79. Ramírez, S. *FastAPI* (MIT, 2018); https://fastapi.tiangolo.com/
80. Consortium for Standardization in Lab Automation Standards. *SiLA Rapid Integration* https://sila-standard.com/standards/ (SILA, 2017).
81. Zhang, L., Merrifield, R., Deguet, A. & Yang, G.-Z. Powering the world's robots—10 years of ROS. *Sci. Robot.* **2**, eaar1868 (2017).
82. Munroe, R. Standards. *xkcd* (2011); https://xkcd.com/927/
83. Volk, A. A. & Abolhasani, M. Performance metrics to unleash the power of self-driving labs in chemistry and materials science. *Nat. Commun.* **15**, 1378 (2024).
84. Volk, A. A. et al. AlphaFlow: autonomous discovery and optimization of multi-step chemistry using a self-driven fluidic lab guided by reinforcement learning. *Nat. Commun.* **14**, 1403 (2023).
85. Snapp, K. L. & Brown, K. A. Driving school for self-driving labs. *Digit. Discov.* **2**, 1620–1629 (2023).
86. Abolhasani, M. & Kumacheva, E. The rise of self-driving labs in chemical and materials sciences. *Nat. Synth.* **2**, 483–492 (2023).
87. Epps, R. W., Volk, A. A., Ibrahim, M. Y. S. & Abolhasani, M. Universal self-driving laboratory for accelerated discovery of materials and molecules. *Chem* **7**, 2541–2545 (2021).
88. Yang, Z. et al. Exploring and unleashing the power of large language models in automated code translation. In *Proc. ACM on Software Engineering* 585–1608 (ACM, 2024).
89. Bran, A. M. et al. Augmenting large language models with chemistry tools. *Nat. Mach. Intell.* **6**, 525–535 (2024).
90. Yoshikawa, N. et al. Large language models for chemistry robotics. *Auton. Robot.* **47**, 1057–1086 (2023).
91. Lunt, A. M. et al. Modular, multi-robot integration of laboratories: an autonomous workflow for solid-state chemistry. *Chem. Sci.* **15**, 2456–2463 (2024).
92. Lunt, A. *sgalunt/Thesis_Amy_Lunt* (GitHub, 2023); https://github.com/sgalunt/Thesis_Amy_Lunt/blob/main/Appendix%204%20ARChemist%20code/Recipe%20Files/yumi_recipe.yaml
93. Clarke, E. *tests/files/orgsyn_v83p0184a.xdl* (GitLab, 2021); https://gitlab.com/croningroup/chemputer/xdl/-/blob/master/tests/files/orgsyn_v83p0184a.xdl
94. *autoprotocol-python* (GitHub, 2023); https://github.com/autoprotocol/autoprotocol-python/tree/master
95. Laboratory of Artificial Intelligence. BioBlocks. *GitHub* https://github.com/liaupm/BioBlocks (2020).
96. Felleisen, M. On the expressive power of programming languages. In *ESOP '90* (ed. Jones, N.) 134–151 (Springer, 1990); https://doi.org/10.1007/3-540-52592-0_60
97. Cunningham, K., Ericson, B. J., Agrawal Bejarano, R. & Guzdial, M. Avoiding the Turing tarpit: learning conversational programming by starting from code's purpose. In *Proc. 2021 CHI Conference on Human Factors in Computing Systems* 1–15 (Association for Computing Machinery, 2021); https://doi.org/10.1145/3411764.3445571
98. Meyer, B. *Object-Oriented Software Construction* (Pearson Education, 2023).
99. Fahland, D. et al. Declarative versus Imperative Process Modeling Languages: the issue of understandability. In *Enterprise, Business-Process and Information Systems Modeling* (eds Halpin, T. et al.) 353–366 (Springer, 2009); https://doi.org/10.1007/978-3-642-01862-6_29
100. Gongora, A. E. et al. A Bayesian experimental autonomous researcher for mechanical design. *Sci. Adv.* **6**, eaaz1708 (2020).
101. MacLeod, B. P., Parlane, F. G. L., Brown, A. K., Hein, J. E. & Berlinguette, C. P. Flexible automation accelerates materials discovery. *Nat. Mater.* **21**, 722–726 (2022).

Acknowledgements

We would like to thank H. Moran (North Carolina State University) for discussions around making the computer science jargon more accessible. M.A. gratefully acknowledges financial support from the University of North Carolina Research Opportunities Initiative (UNC-ROI) and National Science Foundation (award nos. 1940959 and 2208406).

Author contributions

R.B.C. conceived and draughted the work. M.A. and R.B.C. reviewed and edited the manuscript.

Competing interests

The authors declare no competing interests.

Additional information

Supplementary information The online version contains supplementary material available at https://doi.org/10.1038/s44160-024-00649-8.

Correspondence and requests for materials should be addressed to Milad Abolhasani.

Peer review information *Nature Synthesis* thanks Xiaonan Wang and the other, anonymous, reviewer(s) for their contribution to the peer review of this work. Primary Handling Editor: Peter Seavill, in collaboration with the *Nature Synthesis* team.