arXiv:2603.16650v1 [quant-ph] 17 Mar 2026

FAlCon：量子点器件算法控制的统一框架

Tyler J.Kovach [1†⋆], Daniel Schug [2,3,4⋆], Zach D.Merino [4], Mark Friesen [1], Mark A.Eriksson [1], Justyna P.Zwolak [3,4,5‡]

1 美国威斯康星州麦迪逊市威斯康星大学麦迪逊分校物理系，邮编 537062美国马里兰州大学公园市马里兰大学计算机科学系，邮编 207423 美国马里兰州大学公园市马里兰大学量子信息与计算机科学联合中心，邮编 207424 美国马里兰州盖瑟斯堡市美国国家标准与技术研究院，邮编 208995 美国马里兰州大学公园市马里兰大学物理系，邮编 20742

† tkovach@wisc.edu ‡ jpzwolak@nist.gov ⋆这些作者贡献相同。

摘要

随着基于自旋的量子系统规模不断扩大，其搭建与控制的复杂度也急剧上升。在半导体量子点（QD）实验中，器件间差异、异构的控制电子学堆栈以及不同的操作模式，使得在不同实验室之间复用表征、校准和控制逻辑变得困难。我们推出FAl-Con，这是一个开源软件生态系统，旨在提供可移植的自动化表征与调谐测量工作流。FAlCon **提供：(i) 一种轻量级领域专用语言，用于以硬件无关形式表达基于状态的调谐逻辑；(ii) 专用、可传输的物理信息驱动的量子点数据结构库（“调谐术语集”）；以及 (iii) 可扩展的共享测量协议库，从而实现互操作的实验室无关测量技术栈。通过将算法意图与仪器实现分离，同时保留可追溯性并支持类型化脚本编写，**FAlCon **使研究人员和工程师能够在异构量子点设备之间交流、调整和部署表征与自动调谐例程。该框架面向各类用户，既包括在自定义初始条件下运行预构建算法的终端用户，也包括扩展仪器支持并贡献新调谐策略的开发者。尽管当前版本面向量子点实验，其他量子比特模态和科学实验也可以通过提供新的调谐数据类型和仪器控制模板，复用**FAlCon**的模块化抽象。**

目录

1 引言 2

2 动机 4

3 FAlCon框架 5
3.1 falcon-lib 7
3.1.1 领域专用语言 7
3.1.2 运行时引擎 8
3.2 falcon-core 8
3.3 falcon-core-libs 9

3.4 instrument-hub 113.
5 instrument-script-server 113.
6 falcon-measurement-lib 12

4 算法设计 124.1 数据结构定义 134.2 双量子点系统的电荷构型导航 14

5 结论 16

参考文献 17

1 引言

半导体量子点（QD）是基于自旋的量子信息处理领域的领先平台，具有高密度量子比特封装能力，并兼容最先进的半导体制造工艺 [1–6]。随着实验从少量量子点器件扩展到包含数十或数百个量子点的一维和二维（1D 和 2D）扩展阵列，表征器件、校准运行条件并维持稳定性所需的工作量，已成为物理量子比特形成与运行的主要瓶颈。这项运行工作涵盖器件引导、電荷态初始化、隧穿势垒和点间耦合调节、电荷传感器校准和读出配置、量子比特初始化、量子门表征，以及在多个电压子空间和时间尺度上持续进行漂移补偿。在许多实验室中，调谐仍主要依赖人工操作，需要专家介入；随着器件尺寸增大，这种方式变得越来越不切实际。

量子点器件运行面临的核心挑战之一是显著的器件间差异性 [2,4]。异质结构、材料无序和器件架构的差异意味着，没有两个器件会表现出完全相同的响应，也不会具有完全相同的调谐轨迹。因此，在一个样品上成功的调谐流程，转移到新器件时通常需要进行大量调整，即使是在相同材料体系内也是如此。控制电子学堆栈缺乏标准化、测量方式多样，以及稀释制冷机内部布线各不相同，导致软件接口和测量流程存在广泛差异，进一步加剧了这一挑战。consequently，原则上可以共享的高层级表征和调谐逻辑，往往需要针对本地仪器应用程序编程接口（API）和实验室特定脚本约定重新编写，从而限制了跨团队复用，并减缓了整个领域的迭代速度。

这些困难推动了量子点表征和调谐自动化方面的显著进展 [7]。算法和基于规则的方法已证明能够可靠地导航至目标电荷区间，并高效调整工作参数 [8–10]。机器学习方法进一步实现了稳健的状态识别、特征提取，以及在现实测量噪声和器件差异性条件下的原位自动调谐 [11–14]。然而，许多成功的演示仍与特定的实验控制堆栈、数据表示和仪器接口紧密耦合，因此若没有大量重新实现，很难在不同实验室之间交换调谐例程。

在这项工作中，我们介绍了 `FAlCon` （算法控制框架），这是一个开源软件生态系统，旨在解决量子点社区中的可移植性差距。

FAlCon 是一个软件栈，它（1）定义了一种用于编写自动调谐算法的高级领域专用语言（DSL），（2）定义了一个内容丰富的物理信息数据结构库，以及（3）将这些算法连接到与实验室无关的面向仪器接口。 FAlCon使表征、校准和调谐例程能够以硬件无关的形式表达，并部署在异构实验室配置中。

从软件工程的角度来看， FAlCon 采用软件套件而非单体应用程序的形式实现。其基础是一个定义明确的核心库，其中包含可序列化的数据结构，用于表达基于物理的量子点调谐概念。在此库之上，语言绑定使工具和服务能够在多种编程环境中实现，而无需重新实现核心语义。在各个库和应用程序之间，存在一个定义明确的跨语言通信协议，用于发送和接收任何核心库数据结构。这样便可以将它们替换为其他兼容实现，这些实现共同构成一个完全兼容的软件套件。

虽然当前实现面向量子点实验，但 FAlCon的模块化抽象旨在通过为指定领域提供新的数据类型和仪器模板，推广至其他科学应用场景。该框架旨在支持广泛的用户群体，从使用自定义初始条件运行预构建例程的实验人员，到贡献新调谐策略并扩展仪器支持的开发者。随着量子点处理器扩展至更大的阵列，并需要进行大量校准和持续自动重新调谐，这些能力将变得愈发重要。本文接下来将介绍 FAlCon 的架构及其在量子点自动调谐中的应用。我们还将讨论如何将其扩展至其他数据类型和不同的仪器后端。

FAlCon 可从 GitHub 下载，源代码依据 Mozilla公共许可证（MPL）2.0 和 BSD三条款许可证（BSDv3）发布。目前的版本由以下七个公开代码仓库组成，这些代码仓库均位于 FAlCon GitHub 组织 [15]中。

falcon-autotuning

falcon-autotuning 定义了 FAlCon的组织级控制层。

falcon-lib

falcon-lib包含用于将自主控制和优化算法编码为状态机的领域特定语言（C++）、用于解析、解释和部署这些例程的相关运行时工具（C++），以及该语言的初始库（采用 falcon DSL 编写）。

falcon-core

falcon-core 提供了面向性能的基础（主要使用 C++ 并提供 C API），用于支持整个生态系统中使用的规范数据类型和可移植序列化。

falcon-core-libs

falcon-core-libs 通过多语言绑定（目前包括 Python、Go、OCaml 和 Lua）提供 falcon-core 功能，以支持不同编程环境中的工具和集成。

falcon-instrument-hub

`falcon-instrument-hub` 提供了一个抽象接口层（主要使用 Go 编写），用于 falcon 测量请求解析、仪器执行命令请求和测量数据传输。

`instrument-script-server`

`instrument-script-server`实现面向实验室的执行与编排服务（主要使用 C 和 C++ 编写，并支持基于 Lua 的脚本），提供进程隔离的仪器工作进程、基于插件的驱动程序、仪器间同步以及结构化结果捕获。

`falcon-measurement-lib`

`falcon-measurement-lib` 定义了由模式驱动的测量接口层（JSON Schema，包含生成器以及以 Go 和 Lua/Teal 编写的配套代码），用于规范脚本上下文和可复用类型，并支持代码生成，从而实现更安全的脚本开发与验证。

这些代码仓库共同实现了算法意图与仪器实现的分离，同时提供了一条切实可行的途径，可在异构的量子点实验室配置中部署可移植的基于状态的例程。

2 动机

从实验人员的角度来看，自动调谐面临的一项常见挑战，并非高级逻辑的设计，而是如何分发和复用由此产生的代码。在量子点社区中，调谐软件往往与其开发所依托的实验技术栈密不可分。实际上，“自动调谐算法”通常隐含地依赖于本地仪器接口、测量编排、数据格式、校准惯例，以及对采集时序和硬件能力的假设。因此，即使不同配置中的物理调谐目标相似，在一种配置中可靠运行的调谐例程也很难作为即插即用组件在另一种配置中复用。

图 1 展示了当前研究中并存的多种量子点器件设计。这些器件在异质结构、栅极布局和工作模式方面存在显著差异，包括积累型和耗尽型工作模式。图 1(a) 展示了一种积累型器件，其栅电极按功能自然分组（储库、屏蔽、柱塞和势垒）。图1(b) 展示了另一种积累型器件上的高密度栅极架构，其中包含许多可独立寻址的柱塞电极以及多个储库和屏蔽栅极。图 1(c) 展示了一种具有独特光刻布局和布线几何结构的耗尽型器件。尽管存在这些差异，许多调谐工作流仍有共同模式，包括建立稳定的电荷限制、配置电荷传感，以及通过迭代调整栅极电压以达到目标工作点。

然而在实践中，共享算法并不容易，因为可移植性的边界不仅涉及设备本身，还延伸至周边的仪器设备、控制软件和数据基础设施。一段程序会隐含地假设特定的仪器接口和测量原语、特定的编排模型（例如交互式控制还是分阶段序列），以及存储和解读结果的惯例。因此，移植代码往往

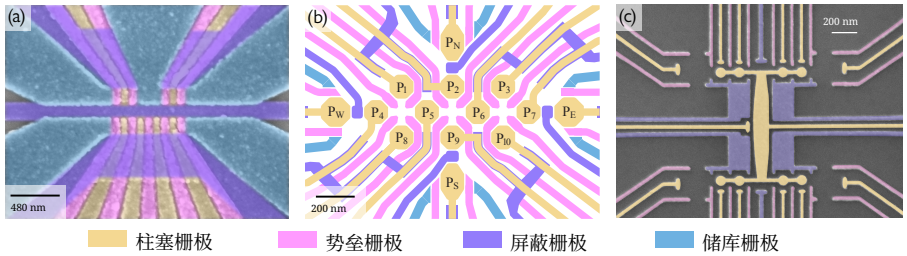


图1：三种栅极定义量子点器件架构示例，展示了布局的多样性以及栅极（储库、屏蔽、柱塞和势垒）的共同功能。在原则上，这些系统均可调谐至能够容纳适用于自旋量子比特运行的量子点的运行状态。(a) 一种硅锗量子点器件 [16], 底部通道中最多可容纳四个量子比特量子点，顶部通道中设有两个电荷传感器。(b) 一种锗量子点器件 [17], 屏蔽栅极之间的中心区域最多可容纳十个量子比特量子点，阵列外侧设有四个电荷传感器。(c) 一种砷化镓量子点器件 [18], 最多可容纳八个可能的量子点，分为四组双量子点，每对旁边设有四个电荷传感器。

归结为重建周边的软件环境，或分叉现有实现并使其适配新的技术栈。文献 [16]和文献 [18]中介绍的两种自举算法，就是这一问题的典型例子。尽管这两种方法的目标都是让设备进入有用的工作状态，但它们的实现无法直接互换，因为二者都与各自不同的实验堆栈和数据表示紧密耦合。其结果是工程工作量重复，且跨平台系统性比较算法的机会减少。

这种可移植性差距不仅影响算法开发者，也影响实验终端用户。终端用户往往希望研究某项受控物理变化（例如异质结构堆栈、栅极几何结构、应变或工作模式）如何影响器件行为，同时保持调谐流程不变。当调谐代码无法复用时，用户就必须投入精力重建或重新发明达到预期运行状态所需的工具，而且通常只实现单一实验配置所必需的功能。这加剧了碎片化，使调谐软件局限于各个实验室，通用性有限，也难以进行基准测试或扩展。

FAlCon旨在解决这一问题，使自动调谐例程能够在异构的量子点实验堆栈之间移植。它让算法开发者能够以硬件无关的形式发布例程，终端用户则只需提供仪器专用模板和配置，即可在不同的实验室配置中运行这些例程。通过将算法意图与仪器实现分离， FAlCon 旨在减少重复的工程工作，提高调谐工作流的可复现性，并支持在不同设备和实验室之间更直接地比较自动调谐策略。

3 FAlCon框架

FAlCon 是一个开源生态系统，由多个相互关联的代码仓库组成，共同支持量子点实验的可移植表征、校准和自动调谐工作流。该生态系统采用模块化设计，使各个组件能够随着新设备、仪器堆栈和用户需求的出现而不断演进。

从较高层面来看， FAlCon 自然划分为一个测量执行服务器和一个 n

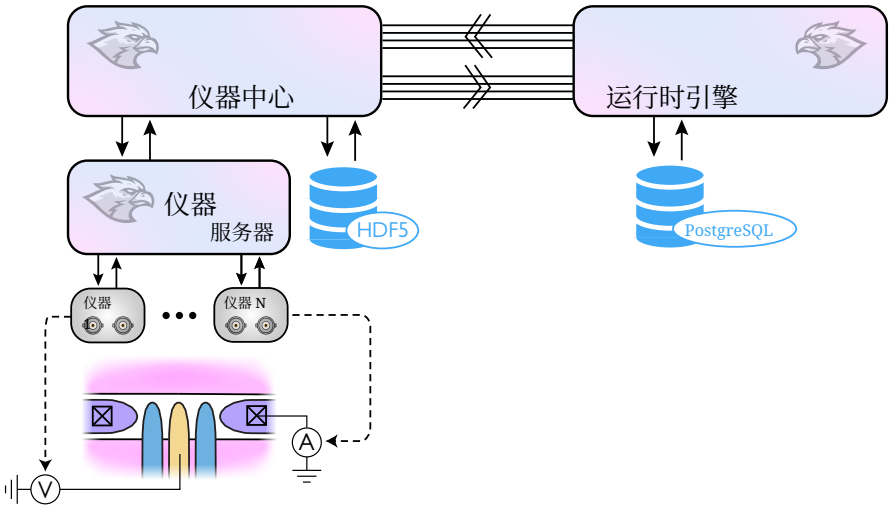


图 2： FAlCon 控制与测量技术栈的高层架构。运行时引擎（右侧）执行 FAlCon 例程，维护调谐状态和全局变量，并将结构化元数据持久化存储在 PostgreSQL 数据库中。测量请求和结果通过消息传递（例如通过 NATS）与实验室侧的测量技术栈交换。仪器中心（左侧）接收来自运行时引擎的测量请求，并将其转换为由仪器服务器执行的具体操作。仪器服务器管理仪器的生命周期，并将请求分发给一个或多个已连接的仪器（仪器 1，...，仪器 N），这些仪器直接连接实验硬件。仪器中心汇总返回的结果，并将测量数据集存储为可移植的 HDF5制品 [19]。底部示意图展示了量子点测量中典型的器件级接线情境：电压源连接到栅极（V），并在欧姆接触处（A）进行电流读出。

算法决策运行时[1]。测量执行服务器负责实验室层面的仪器控制、时序控制和数据采集。算法决策运行时维护算法状态、解读结果，并选择调谐例程中的下一步操作。这两个组成部分进行双向通信，以便根据最新测量结果调整算法决策，并使测量执行能够向算法层反馈状态、故障和元数据。

这种划分源于低噪声低温实验的实际限制，以及数据驱动分析计算成本的不断增加。现代自动调谐工作流通常会集成机器学习模型，用于解读测量结果并指导后续操作 [7, 13]。模型推理及相关预处理可能需要大量 CPU 和内存资源，有时还可受益于 GPU加速。在实践中，将繁重的计算集中在测量计算机上通常并不可取，既不利于最大限度地减少本地电磁干扰，也不利于保持采集栈轻量且可靠。 FAlCon 这种架构支持在与测量服务分离的硬件上运行算法运行时，该硬件甚至可以位于不同的物理位置，同时通过消息传递维持低延迟协调。在我们的参考实现中，这种协调通过 NATS消息系统实现 [20]。图2 概述了这种分解方式及主要通信路径。

该 FAlCon 生态系统以 GitHub 组织的形式分发，作为公开 ic

[1]下文及以后 运行时是指程序正在执 行的期间。

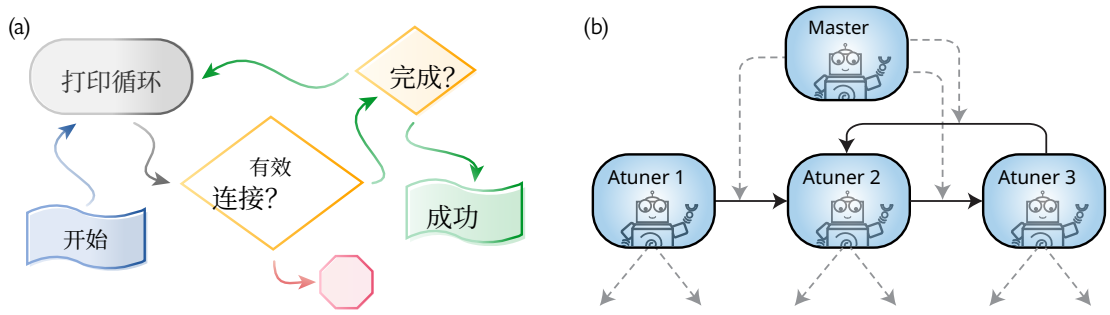


图 3: **(a)** 一个简单的 FAlCon 领域特定语言程序以信号流图形式可视化的示例。在此示例中，程序遍历连接列表，打印每个有效连接，并在列表耗尽时终止。该 FAlCon程序的实现见清单5。 **(b)** 中层次化组合的示意图 FAlCon。父级（Master）自动调谐器将多个子级自动调谐器（Atuner 1–3）嵌套为内部状态，从而可以将现有自动调谐器组合成更高级别的工作流，实现复用。

入口和当前开源代码库的集合 [15]。该组织的主页概述了项目目标、开发状态、社区期望，并提供了指向主要代码仓库的链接。在本文其余部分中，我们用“FAlCon”指代整个测量与控制生态系统，而非任何单个代码仓库。在以下小节中，我们将介绍当前版本中的主要代码仓库及其在整体框架中的作用。

3.1 falcon-lib

从概念上讲， falcon-lib 将控制平面层（用于表示并执行调谐逻辑）与测量平面层（用于连接实验室仪器并采集数据）分离开来。控制平面以面向用户的领域特定语言（DSL）为核心，用于将调谐例程表示为层次化的基于状态的程序。相应的运行时引擎会解析并执行 DSL 程序，同时向外部服务和分析库提供受控接口。在参考实现中，控制服务与测量服务之间的双向通信通过 NATS 等消息传递协议实现 [20]。外部通信主要通过结构高度规范化的 PostgreSQL 数据库绑定处理；参见图2。这种分离使计算密集型分析和决策能够在专用硬件上执行，同时让测量计算机在实验室环境中保持轻量且稳定。以下各小节将介绍 DSL 及其运行时模型的关键抽象。

3.1.1 领域特定语言

FAlCon 的核心设计目标之一，是将自动调谐例程表示为显式的基于状态的程序，使其能够组合、复用并可重复执行。许多量子点自动调谐工作流都可以表示为类似流程图的结构，其中包含条件分支，并在测量与分析步骤之间迭代 [16]。 FAlCon 通过一种旨在支持层次化组合的领域特定语言（DSL），将这种结构形式化。从较高层次来看，由此形成的程序可自然地映射为信号流图，其中节点表示操作或子程序，边则编码状态转换和控制流 [21]。

图 3(a) 展示了一个最简示例：程序遍历一个连接列表，并

打印每个有效条目。流程图中的节点表示程序状态[2](例如， `Start`， `Print Loop`) 和操作（例如， `Valid Connection?`， `Done?`），而有向边则表示状态转换和条件分支。尽管这个最简示例有意做得较为简单，但它体现了实际调谐例程中相同的结构。例如，可以将“打印循环”替换为一个例程，用于采集电荷稳定性图、评估质量指标（例如传感器峰可见度），并根据该指标是否超过阈值进行分支。如果指标低于阈值，程序便更新工作点，并重复采集和评估。如果指标超过阈值，程序便转换到成功状态，并进入工作流的下一阶段。

不过，采用专用领域特定语言（DSL）的主要动机并非支持状态机结构——通用语言通常也具备这些结构——而是将分层组合作为一等特性来支持。图3（b）展示了这种嵌套关系：外层自动调谐器（`Master`）定义了一组状态，而这些状态的实现本身也是自动调谐器（`Atuner` 1–3），并指定了各状态如何转换到下一个状态。这种分层系统只需导入现有调谐算法，将其定义为状态，并指定它们如何在更高层级转换到其他状态，即可实现复用。

3.1.2 运行时引擎

运行时引擎通过解析源文件、构建内部程序表示，并在执行过程中解释状态转换，来执行 `FAlCon` 领域特定语言（DSL）程序。该实现由 GNU Bison 解析器 [22], 负责处理，旨在清晰地区分解析、程序表示和执行。因此，今后更新语言时，只需修改解析器文件即可实现。

在我们的参考实现中，运行时引擎支持外部函数接口（FFI），使例程能够使用通过 C API 提供的外部库。在 DSL 中实现状态逻辑，同时依赖对性能敏感的分析代码或直接测量控制库时，这种 FFI 设计尤其有用。在当前架构中，与测量系统的交互是通过这些外部接口实现的，而不是将仪器控制代码直接嵌入 DSL。不过，也可以通过 FFI 直接绑定常用的分析算法，以供未来的原生 `FAlCon` DSL 代码使用。

3.2 `falcon-core`

`falcon-core` 提供了整个`FAlCon` 生态系统所使用的基础数据类型和底层实用工具。它使用C++14实现，旨在支持量子点器件的高性能自动调谐工作流。该代码库围绕 C++ 核心（`cpp/`）以及 C API 封装层（`c-api/`）组织，后者旨在支持语言绑定和外部工具。一个重要的设计目标是为常见实验概念提供规范表示，从而实现依赖应用程序之间一致的数据交换。

`falcon-core`包含大量量子点调谐相关数据类型。一个示例是 `Connection`，如清单1所示，它表示真实器件接触结构中某个成员的命名别名及其功能角色（此处为势垒栅）。

数据类型的一项关键属性是 `falcon-core` 它们能够进行序列化。所有数据类型都通过继承通用的 `Song` 类来获得序列化能力。该虚基类实现了 `from_json_string`方法，可直接从 JSON 构建每种类型；还实现了`to_json_string` 方法，可直接将每种类型转换为字符串。 `Song` 的实现采用了 `cereal` 库 [23]。直接由继承自 Song 的类型组合属性，可确保整个对象都能序列化。

[2]状态指调优算法中的一个步骤，在此步骤中，自动调谐器会执行某项操作（例如测量或分析），或等待信息，然后再转换到另一个状态。

```
namespace falcon_core::physics::device_structures {
e
c
```

清单 1： `falcon-core`中 `Connection` 数据类型的核心定义。AConnection hasa nameand a type (`DeviceFeature`) specifying its functional role.

第二个关键特性是，这些类型专为并发使用而设计。示例实现使用共享互斥锁[3] 在序列化期间保护内部状态；参见清单 2。

为了让这些数据类型能够在其他编程环境中使用， `falcon-core`提供了功能完备的兼容 C99 的 API，以及 C++ 库。此 C API 构成一种沙漏模式[4]，，其中 C 句柄管理由 C++层拥有的对象。清单3展示了 `Connection` 数据类型的这种沙漏模式。若要扩展这些数据类型，开发者可以向 `falcon-core` 添加新的数据类型，也可以构建一个继承 `Song`属性的新库。

3.3 `falcon-core-libs`

`falcon-core-libs` 为 `falcon-core`中实现的规范数据类型提供多语言绑定。这些绑定基于 `falcon-core`公开的兼容 C99 的 API 构建。该设计使使用高级语言编写的工具和服务能够构造、检查，

[3]互斥锁（mutual-exclusion lock）是一种同步原语，通过确保同一时间只有一个线程能够访问或修改共享数据，来防止竞态条件。[4]沙漏模式是一种编写精简 C API 的方式 [24]。

```
class Connection : public virtual generic::Song {
```

清单 2： `Connection`的线程安全序列化。该实现使用共享互斥锁在序列化过程中保护内部状态。

并序列化 `falcon-core` 数据类型，无需重新实现其语义或重复编写序列化逻辑。

该代码仓库目前包含面向多种编程环境的绑定，包括 Go、Python、OCaml 和 Lua。其总体目标是让生态系统的不同部分能够使用最适合其用途的语言实现，同时为测量类型和结果保留共享的核心表示。为支持底层 C API 的演进，`FAlCon` 其组织方式使得当 `falcon-core-libs` 类型和函数发生变化时，可以重新生成 `falcon-core` 特定于语言的绑定[5]。

[5]C API 暴露的函数自然可分为分配、释放以及读/写访问等类别。这些类别为绑定生成器提供了便利的基础，并有助于确保内存所有权

```
extern "C" {typedef void* ConnectionHandle;
```

清单 3： `falcon-core` 中 `Connection` 数据类型的 C API 库定义。每个方法都通过采用 C++ 对象类型并附加相应的方法进行名称修饰。C API 维护对底层 C ++ `falcon-core`库所使用的共享指针的句柄。

3.4 instrument-hub

该 instrument-hub 在 FAlCon架构中充当中央注册表和发现服务，提供统一接口，用于定位、查询和寻址已连接的仪器；参见图 2。通过 instrument-script-server （详见 3.5）实例化仪器后，该仪器便会注册到 instrument-hub中，并公开描述该仪器、其可用参数以及它能够生成或使用的测量量的元数据。

在测量执行过程中， instrument-script-server 会在计划测量任务中生成测量结果，并将其近实时转发至 instrument-hub 。 instrument-hub会汇总这些结果、管理缓冲，并将生成的数据集持久化到 HDF5 数据存储 [19]。

instrument-hub 的一项关键设计原则是系统组件之间松耦合。仪器在初始化时无需相互了解；相反，执行测量例程时会通过 instrument-hub 动态解析仪器引用。这使实验配置能够在运行时进行组合、重新配置和扩展，而无需重启控制系统。例如，锁相放大器和磁体电源驱动程序可以独立初始化，之后在扫描测量中协同使用，由 instrument-hub 解析测量工作流所需的仪器引用。

3.5 instrument-script-server

instrument-script-server提供 FAlCon的实验室专用仪器执行与编排服务。它旨在将测量例程作为脚本运行，这些脚本引用仪器配置，同时将面向硬件的代码与更高层的调谐逻辑隔离开来。服务器执行用户提供的 Lua 脚本，使实验逻辑能够打包、进行版本管理，并作为结构化测量任务重放。这种面向任务的执行模式支持脚本及其所需配置和依赖项的暂存与部署，并为测量的运行和监控提供定义明确的生命周期。instrument-script-server管理每个仪器驱动程序的完整生命周期，包括实例化、初始化（例如，打开通往物理硬件接口的 VISA/串行/TCP连接）、正常运行和拆卸。为提高稳健性，系统通过在单独的工作进程中运行每个仪器（或驱动程序实例）来支持进程隔离，从而降低单个故障驱动程序导致整个测量会话不稳定的风险。

该 instrument-script-server与其他常用测量系统的行为不同，例如 QCoDeS 和 Labber，这些系统强制采用面向用户的单线程 API。为了与仪器通信，
instrument-script-server 会运行 Lua 脚本，通过配置好的驱动程序接口发送仪器原生命令来执行实验。

每个 instrument-script-server 实例都作为网络服务运行，通过标准网络协议（例如 TCP/IP ）与其他 FAlCon 组件通信。这使得服务器可以部署在一台专用的实验室计算机上，并与仪器进行物理连接，同时实验人员则可通过另一台工作站或远程位置与系统交互。

除单仪器操作外， instrument-script-server 还内置支持协调式多仪器测量。服务器支持并行命令执行和同步原语，使脚本能够协调各仪器的操作并保持一致的时序关系。结果和执行元数据将

并且对象生命周期在不同语言边界之间保持明确定义。

通过服务器的集成接口返回给上游使用者，作为可追溯测量记录的一部分。

一种常见的测量模式是参数扫描，即在指定范围内逐步调整一个或多个自变量，并在每个点记录一个或多个因变量。在 `instrument-script-server` 中，这类测量模式以脚本实现，并受益于集中式编排、并行性，以及参与测量的仪器之间的同步。由于执行层采用集中式架构，脚本无需重新实现底层驱动程序管理，也无需编写临时的多仪器协调逻辑。

3.6 `falcon-measurement-lib`

`falcon-measurement-lib` 提供 FAlCon的模式驱动测量接口层。其核心目标是标准化异构实验室配置中测量请求、脚本上下文和返回数据产品的表示方式。它定义了规范的 JSON 模式[6]，用于描述可复用类型（例如仪器目标和域），以及各脚本专属的模式，用来指定测量脚本所需的全局变量和可用上下文。这种模式优先方法能够验证测量请求，并通过明确规定测量接口（而非将其隐含在临时脚本中）提升可移植性。

除模式定义外，`falcon-measurement-lib` 还提供代码生成工具，可自动生成编辑器类型提示和特定语言的类型定义等辅助文件。这些生成的输出包括用于编辑器集成的 Emmy 风格类型信息、用于编译器和工具支持的 Go 结构体类型，以及用于支持类型化脚本工作流的可选 Teal 脚手架。 `falcon-measurement-lib` 还提供 Lua 运行时辅助模块，用于实现模式引用的功能并支持脚本执行。这些组件共同确保 FAlCon生态系统中的测量接口保持一致，并在扩展测量库或添加新脚本时减少集成错误。

`falcon-measurement-lib`定义测量模板，以便在 FAlCon 生态系统中实现可移植的数据采集。库中的每个模式都定义了模板化的测量样式。以下 `set-get` 清单 4展示了一个测量示例。此示例体现了一种常见模式：设置一个或多个仪器参数，然后采集并返回一个或多个值。这些模板提供了一套可移植的词汇，用于表达测量意图；借助其余的 FAlCon 技术栈，可针对特定实验室的仪器配置对其进行编译或执行。

4 算法设计

算法以 FAlCon 领域特定语言 表达。该领域特定语言将调谐例程表示为基于状态的程序，包含命名状态、类型化输入和输出，以及定义明确的转换逻辑。这种设计与许多 QD 调谐工作流的主要结构相契合；这些工作流通常可以自然地分解为包含条件分支、重试和终止条件的分阶段流程。该领域特定语言将调谐意图与实验室特定软件细节隔离开来，降低了偶然复杂性，并使例程更易于在不同团队间交流。它还提供了一个受控的扩展接口，使 FAlCon 能够随着需求变化而演进语言特性和执行后端。

[6]JSON Schema 是一种开放且获得 IETF 认可的标准语言，以声明方式定义 JSON 数据的结构和内容，用于验证、文档编写和交互控制。

```
{    "title": "Measure_Get_Set",    "properties": {      "setVoltages": {
vol
sam
of
      "description": "List of MeasurementResponse with a numeric value in nA.",      }
  }
}
```

清单 4：用于执行直流 get/set 操作的脚本代码片段。它支持在指定仪器上设置电压，并从其他仪器读取数值，还支持批量操作、并行执行和日志记录。此上下文通常用于需要协调设置输出并从仪器采集数据点的脚本。

4.1 数据结构定义

清单 5提供了一个简洁且主要用于教学的示例，展示了 FAlCon领域特定语言的构造以及全文采用的状态机编程模型 FAlCon。程序以一个 import 代码块开头，用于加载所需的库（第 1 行）。仅支持 FAlCon 原生库。在第 2 至 10 行中，定义了一个新的数据结构 DeviceConnection 。 DeviceConnection 包含一个名为 name_ 的属性，以及两个相关例程：一个名为 New 的方法和一个用于访问该属性的例程 Name。第 11 至 39 行定义了一个自定义 autotuner，CheckPlungerGates它对设备连接列表进行简单检查，并展示了状态机模型中的条件分支和错误处理。 CheckPlungerGates 这是一个有限状态程序示例，具有类型化输入和输出、局部状态变量（index、 connection和 err），以及一个例

```
import ("log" "io" "config" "array")
struct DeviceConnection {
    string name_;
    routine New(string name) -> (DeviceConnection conn) {
        name_ = name;
    }
    routine Name() -> (string name) {
        name = this.name_;
    }
}
autotuner CheckPlungerGates (array::Array<DeviceConnection> connections,
                             config::Config conf) -> (Error err) {
    int index = 0;
    err = nil;
    DeviceConnection connection = connections[index];
    start -> loop (connection);
    state loop (DeviceConnection conn) {
        io::println("DeviceConnection name: " + conn.Name());
        if (!conf.GetPlungerGates().Contains(conn)) {
            log::warn("DeviceConnection " + conn.Name() +
                      " is not in plunger gates.");
           -> missing_plunger_gate;
        } elif (index < connections.Size()) {
            index = index + 1;
           -> loop(connections[index]);
        } else {
           -> exit;
        }
    }
    state missing_plunger_gate (DeviceConnection conn) {
        err = Error("DeviceConnection " + conn.Name() +
                    " is missing a plunger gate.");
       -> exit;
    }
    state exit {
        log::info("Finished processing connections.");
        terminal;
    }
}
```

列表5：示例展示了核心 FAlCon 领域特定语言构造，包括导入、用户定义的数据类型，以及一个 autotuner ，它以基于状态的程序形式表达，并包含显式状态转换和错误处理。

显式开始状态转换进入主 loop 状态。在 loop中，自动调谐器遍历设备连接，打印每个条目，并根据条件逻辑进行分支：要么转换到missing_plunger_gate 以构造并传播错误，要么推进循环以处理下一个连接（第17–29行）。该例程在一个显式的 exit 状态中终止，并标记为 terminal。

4.2 双量子点系统的电荷构型导航

清单6展示了一个更实用的示例，其中给出了一个用于执行电荷构型导航的简单程序。该程序 ChargeConfigurationTuner 定义了一个简单但正确的自动调谐器，用于导航至电荷稳定性空间中的 (n,m) 构型，并隐含地假设在对应于扫描栅极的两个笛卡尔维度中对双量子点系统进行导航。

从算法上看， ChargeCongfigurationTuner等价于一个简单的计数器，并使用一个方向变量以及当前的n和m 变量来导航至一个隐式的terminal 状态。

程序以一个 import 块开始，用于加载所需的 stateStepper、 deviceState和 hub 库。 stateStepper 库实现了一个 BlipStateStepper 自动调谐器，它按照指定的 direction 扫描量子点器件柱塞栅极的电压，以检测电导中的第一个峰值（一个突跳）。随后，它将量子点器件置于该突跳之后的电荷构型单元中心。 right（up）方向表示沿栅极电压扫描的向右（向上）方向检测突跳，这通常分别对应电荷稳定性图中较快（较慢）的扫描轴。 deviceStates 库实现了DeviceStates 数据类型，用于描述整个量子点器件当前的电压配置。该配置进一步分解为若干组成部分，包括一个 DeviceState列表，其中每个DeviceState 都包含对关联 Connection[7] （此处为 PlungerGate）的引用，以及一个 Quantity ，用于表示施加到该栅极的电压，其值为浮点数，并带有一个 SymbolUnit 指示测量单位（此处为伏特）。最后， hub库实现了与instrument-script-server 的绑定，并允许自动调谐器请求当前的 DeviceState

[7]参见列表1 了解定义。

```
1 import ("stateStepper" "deviceStates" "hub")
2 // (n,m) indicates the x,y axis count
3
4   // this autotuner starts the device tuned at (0,0)
5     int current_n=0;
6
8
10
11
13
15
17
20
22
24
26
```

清单 6：一个能够改变量子点器件电荷态的自动调谐器示例。由于该示例并非根据通用器件配置文件定义，因此比较简单。该自动调谐器使用 BlipStateStepper 例程，可确保沿指定方向步进一个电荷态。

直接使用例程 CollectCurrentDeviceState 从仪器获取。

该 ChargeConfigurationTuner 接收两个整数作为输入：m和n，用于描述用户指定的系统目标电荷配置（第 3 行）。自动调谐器到达目标电荷配置后即终止，并返回系统当前的 deviceStates 。初始时， autotuner 假定器件已调谐至 $(0,0)$ 电荷配置。此假设通过两个局部 autotuner 状态变量进行编码，分别称为 current_m 和current_n （第 5–6 行）。这些变量用于统计两个基数测量方向上的脉冲数，默认起始方向设为 up （第 8 行）。

初始配置完成后， ChargeConfigurationTuner状态从 start变为 tuning，并将初始测量方向传递给在 tuning状态内部定义的局部状态变量，再赋给变量名 direction。进入 tuning后， ChargeConfigurationTuner会调用一个 stateStepper::BlipStateStepper 子程序，驱动量子点器件沿传入该子程序的方向移动到相邻单元（第 11 行）。

随后递增 autotuner 状态变量 current_n ，并将其与该方向的目标电荷配置进行比较（第 13–14 行）。如果未达到目标，当前 tuning 状态会沿 up 方向转换为新的 tuning 状态（第 15 行）。否则，当前 tuning 状态会沿 right 方向转换为新的 tuning 状态（第 16–18 行）。沿 right 方向的测量会持续进行，直到 autotuner 状态变量 current_m与目标 m autotuner 状态变量一致。

此时， ChargeConfigurationTuner 会调用 hub 库中的CollectCurrentDeviceState 子例程，以构建更新后的 DeviceStates （第 24 行）。随后， DeviceStates 会存储到名为 dstates 的 autotuner 状态变量中；成功完成 autotuner 执行后，该变量预期会作为输出返回（第 25 行）。

5 结论

随着量子点阵列的规模和复杂度不断增加，可复现、自动化且便于移植的控制工作流变得愈发重要。要实现量子点系统的大规模高效表征与运行，很快就需要通用的自动化控制例程，以便共享、调整和复用。 FAlCon通过标准化数据表示、支持带类型且经过验证的脚本编写，并提供可扩展的架构，以适配设备布局、测量模式和仪器后端的扩展，为构建此类工作流奠定了基础。

FAlCon是一个开源软件生态系统，旨在让表征和自动调谐例程更易于在异构设备平台、控制电子设备和量子比特类型之间共享和部署。它通过将算法意图与仪器实现分离，克服可移植性障碍；同时结合用于调谐逻辑的基于状态的领域特定语言，以及标准化的物理信息数据结构和模式驱动的测量接口。这些组件共同实现了经过验证的类型化脚本编写，并提供了一种在不同设备和实验室之间一致地表示测量请求、上下文和数据产品的方式。

FAlCon 生态系统通过其领域特定语言实现了富有表现力的基于状态的调谐逻辑。将基于状态的调谐逻辑与测量控制后端分离，使这种方法不依赖特定硬件。这种分离得益于一个通用库，其中包含可通信的物理信息量子点数据类型。测量技术栈利用共享测量协议库，可靠地控制完整的实验装置。总的来说，这些设计选择使FAlCon 能够与各种实验室配置中的仪器交互。作为一个不断发展的全栈生态系统， FAlCon 旨在弥合系统工程与实验部署之间的差距，推动最先进的量子点研究。

展望未来，我们期待社区贡献沿着多个 al

方向拓展这一生态系统。近期的扩展包括增加方言类型和测量模板、扩大仪器支持范围，以及建立可在不同设备上进行基准测试的可复用自动调谐器库。构建并共享明确的基准测试模拟，将有助于直接比较算法的性能与效率，不受仪器导致的瓶颈影响，从而为最终用户提供性能最佳的可用算法。更具雄心的方向包括整合模拟与数字孪生工作流，以便离线调试和评估调谐策略；将机器学习决策层作为可选组件加入调谐例程；以及探索将 FAlCon测量命令直接编译为仪器硬件指令集架构——绕过当前在instrument-script-server中使用的手动 Lua 脚本层。随着控制电子设备从分布式独立实验室仪器转向可扩展的片上解决方案，还可以探索将 FAlCon 引擎直接编译到现场可编程门阵列（FPGA）上。

作为一个开源软件项目， FAlCon 在量子物理研究与软件算法开发之间建立了活跃的跨学科联系。通过为算法控制提供共享的软件基础， FAlCon 旨在加快迭代、提升可复现性，并促进不同实验室更系统地比较调谐策略。我们预计，这些能力将有助于减少重复的工程工作量，并推动向规模更大、自治程度更高的基于量子点的量子处理器过渡。

除在半导体自旋量子比特领域的直接应用外， FAlCon引入的抽象旨在推广至其他科学平台，从而支持跨领域实现互操作自动调谐的更广泛愿景。该生态系统旨在服务专业知识水平各异的用户，包括使用自定义初始条件运行预构建算法的用户、开发新自动调谐例程的贡献者，以及扩展仪器支持并升级平台本身的开发者。用户只需将问题简化为嵌套状态机即可。

致谢

资助信息 本研究部分由美国陆军研究办公室资助，资助编号为 W911NF-17-1-0274、W911NF-23-1-0110 和 W911NF-24-2-0043。Z.M. 在美国国家标准与技术研究院获得美国国家研究委员会研究员计划资助期间，参与了本研究的部分工作。本文所述观点和结论仅代表作者意见，不应被视为美国政府明示或暗示的官方政策。尽管本文注明了任何版权信息，美国政府仍有权为政府用途复制和分发本文的再版。提及商业产品仅供参考，并不意味着美国国家标准与技术研究院对其予以推荐或认可。

软件可用性 FAlCon的二进制文件和安装文档可通过 FAlConGitHub 组织获取。如果您觉得这款软件有用，请为相应的代码仓库点星，并引用本文。

参考文献

[1] D. Loss 和 D. P. DiVincenzo，量子点量子计算，《物理评论 A》**57**(1),120(1998), doi：10.1103/PhysRevA.57.120。[2] L. M. K. Vandersypen、H. Bluhm、J. S. Clarke、A. S. Dzurak、R. Ishihara、A. Morello、D. J. Reilly、L. R. Schreiber 和 M. Veldhorst，量子点和施主中的自旋量子比特互连——高温、高密度且相干，

npj Quantum Inf. 3(1), 34（2017年），doi:10.1038/s41534-017-0038-y.[3] L. M. K. Vandersypen 和 M. A. Eriksson，半导体自旋量子计算，Phys. Today 72(8), 38（2019年），doi:doi:10.1063/PT.3.4270.[4] G. Burkard、T. D. Ladd、A. Pan、J. M. Nichol 和 J. R. Petta，半导体自旋量子比特，Rev. Mod. Phys. 95(2), 025003（2023年），doi:10.1103/RevModPhys.95.025003.[5] H. C. George、M. T. M.adzik、E. M. Henry、A. J. Wagner、M. M. Islam、F. Borjans、E. J. Connors、J. Corrigan、M. Curry、M. K. Harper、D. Keith、L. Lampert 等，制造于300毫米半导体制造生产线上的12自旋量子比特阵列，Nano Lett. 第793–799页（2024年），doi:10.1021/acs.nanolett.4c05205.[6] P. Steinacker、N. D. Stuyck、W. H. Lim、T. Tanttu、M. Feng、S. Serrano、A. Nickl、M. Can- dido、J. D. Cifuentes、E. Vahapoglu、S. K. Bartee、F. E. Hudson 等，保真度超过99%的兼容工业生产的硅自旋量子比特单元，Nature 646(8083), 81（2025年），doi:10.1038/s41586-025-09531-9.[7] J. P. Zwolak 和 J. M. Taylor，专题综述：量子点器件控制自动化的进展，Rev. Mod. Phys. 95(1), 011006（2023年），doi:10.1103/RevModPhys.95.011006.[8] T. A. Baart、P. T. Eendebak、C. Reichl、W. Wegscheider 和 L. M. K. Vandersypen，利用计算机自动调谐半导体双量子点至单电子状态，Appl. Phys. Lett. 108(21), 213104（2016年），doi:10.1063/1.4952624.[9]C. J. van Diepen、P. T. Eendebak、B. T. Buijtendorp、U. Mukhopadhyay、T. Fujita、C. Reichl、W. Wegscheider 和 L. M. K. Vandersypen，双量子点中点间隧穿耦合的自动调谐，Appl. Phys. Lett. 113(3), 033101（2018年），doi:10.1063/1.5031034.[10] M. Lapointe-Major、O. Germain、J. Camirand Lemyre、D. Lachance-Quirion、S. Ro- chette、F. Camirand Lemyre 和 M. Pioro-Ladrière，将量子点自动调谐至单电子状态的算法，《物理评论 B》102(8), 085301（2020年），doi:10.1103/PhysRevB.102.085301.[11] S. S. Kalantre、J. P. Zwolak、S. Ragole、X. Wu、N. M. Zimmerman、M. D. Stewart 和 J. M. Taylor，量子点状态识别与自动调谐的机器学习技术，npj Quantum Inf. 5(1), 6（2019年），doi:10.1038/s41534-018-0118-7.[12]J. Ziegler、T. McJunkin、E. S. Joseph、S. S. Kalantre、B. Harpt、D. E. Savage、M. G. Lagally、M. A. Eriksson、J. M. Taylor 和 J. P. Zwolak，迈向含噪量子点器件的稳健自动调谐，《物理评论：应用》 17(2), 024069（2022年），doi:10.1103/PhysRevApplied.17.024069.[13] J. P. Zwolak、T. McJunkin、S. S. Kalantre、J. Dodson、E. R. MacQuarrie、D. Savage、M. Lagally、S. Coppersmith、M. A. Eriksson 和 J. M. Taylor，利用机器学习原位自动调谐双量子点器件，《物理评论：应用》13(3), 034075（2020年），doi:10.1103/PhysRevApplied.13.034075.[14] R. Durrer、B. Kratochwil、J. Koski、A. Landig、C. Reichl、W. Wegscheider、T. Ihn 和 E. Gre- plova，使用神经网络将双量子点自动调谐至特定电荷态，《物理评论：应用》13(5), 054019（2020年），doi:10.1103/PhysRevApplied.13.054019.[15] T. J. Kovach、D. Schug、Z. Merino 和 J. P. Zwolak，`FAlCon`：量子点量子比特自动调谐，GitHub（2026年）。

[16] T. J. Kovach、D. Schug、M. A. Wolfe、E. R. MacQuarrie、P. J. Walsh、O. M. Eskandari、J. Ben- son、M. Friesen、M. A. Eriksson 和 J. P. Zwolak，《自举、自主测试及用于 Si/Si_xGe_{1-x} 多量子点器件的初始化系统》，《物理评论：应用》25(1)，014043（2026年），doi:10.1103/vbtg-fws9.[17] L. E. A. Stehouwer、C. X. Yu、B. van Straaten、A. Tosato、V. John、D. D. Esposti、A. Elsayed、D. Costa、S. D. Oosterhout、N. W. Hendrickx、M. Veldhorst、F. Borsoi 等，《利用应变外延锗，在微米尺度扩展低噪声自旋量子比特》，《自然·材料》页码 1–7（2025年），doi:10.1038/s41563-025-02276-w.[18] A. Zubchenko、D. Middlebrooks、T. Rasmussen、L. Lausen、F. Kuemmeth、A. Chatterjee 和 J. P. Zwolak，《量子点器件的自主自举》，《物理评论：应用》23(1)，014072（2025年），doi:10.1103/PhysRevApplied.23.014072.[19] HDF 集团，《层次数据格式第 5 版》，GitHub（2025年）。[20] NATS 作者，《NATS：适用于自适应边缘系统和分布式系统的连接技术》，GitHub（2017年）。[21] S. J. Mason，《反馈理论——信号流图的一些性质》，《IRE 会议录》41(9)，1144（1953），doi:10.1109/JRPROC.1953.274449.[22] 自由软件基金会，《GNU Bison：Yacc 兼容解析器生成器》，自由软件基金会，马萨诸塞州波士顿，版本 3.8.2，https://www.gnu.org/software/bison/（2024年）。[23] W. S. Grant 和 R. Voorhies，《cereal——用于序列化的 C*++11*库》，GitHub（2017年）。[24] S. D. Toit，《用于 c++ API 的 Hourglass 接口》，GitHub（2014年）。