arXiv:submit/0800979 [quant-ph] 13 Sep 2013

An Integrated Development Environment for Adiabatic Quantum Programming

T S Humble, A J McCaskey†, R S Bennink, J J Billings, E F D'Azevedo, B D Sullivan‡, C F Klymko§, H Seddiqi‖

Quantum Computing Institute, Oak Ridge National Laboratory, Oak Ridge, TN USA

E-mail: humblets@ornl.gov

Abstract. Adiabatic quantum computing is a promising route to the computational power afforded by quantum information processing. The recent availability of adiabatic hardware raises the question of how well quantum programs perform. Benchmarking behavior is challenging since the multiple steps to synthesize an adiabatic quantum program are highly tunable. We present an integrated development environment for adiabatic quantum programming called JADE that provides control over all the steps taken during program synthesis. JADE captures the workflow needed to rigorously benchmark performance while also allowing a variety of problem types, programming techniques, and processor configurations. We have also integrated JADE with a quantum simulation engine that enables program profiling using numerical calculation. The computational engine supports plug-ins for simulation methodologies tailored to various metrics and computing resources. We present the design, integration, and deployment of JADE and discuss its potential use for benchmarking adiabatic quantum programs by the quantum computer science community.

† Present address: Department of Physics, Virginia Tech, Blacksburg, VA USA
‡ Present address: Department of Computer Science, North Carolina State University, Raleigh, NC USA
§ Present address: Department of Mathematics and Computer Science, Emory University, Atlanta, GA USA
‖ Present address: Department of Physics, Georgia Southern University, Statesboro, GA USA

Contents

1 Introduction **2**

2 Adiabatic Quantum Programming **4**
2.1 Quantum Computational Model 4
2.2 Adiabatic Quantum Optimization 5
2.3 Quadratic Unconstrained Binary Optimization 5
2.4 Hardware Embedding 6
2.5 Hardware Schedules and Program Execution 8
2.6 Computational Readout and Problem Solution 9

3 Jade Adiabatic Development Environment **9**
3.1 Use Cases 10
3.2 System Context 11
3.3 Component Architecture 11
3.4 JadeD 12
3.5 NiCE 16
3.6 Sapphire 17
3.7 Simulation Plug-ins 18
3.8 Testing Framework 20

4 Benchmark Demonstrations **21**

5 Discussion **24**

1. Introduction

The discovery of quantum algorithms with significant speed-ups over their classical counterparts has spurred interest in the research and development of quantum computing systems. Several different but computationally equivalent models for quantum computing have emerged including the model of adiabatic quantum computing (AQC) [1]. Notionally, the AQC model for universal quantum computation corresponds to adiabatic (i.e., slow) changes in the state of a quantum physical system. While computationally equivalent to other models, AQC promises some intrinsic benefits for ensuring fault-tolerant computation and reducing system complexity [2].

Additional attention to the AQC model has been stimulated by the recent commercial realization of a special purpose processor that implements the adiabatic quantum optimization (AQO) algorithm [3]. The processor, manufactured by the Canadian firm D-Wave Systems, Inc., realizes a programmable Ising spin-glass model in a transverse field [4]. Although the available hardware is not capable of universal quantum computation, it does provide the first complete realization of a quantum information processing system. Moreover, because the AQO algorithm is broadly applicable to combinatorial optimization problems, it has garnered attention for use in a number of application domains. Examples include problems in classification [5, 6], machine learning [7], graph theory [8, 9, 10], artificial neural networks [11], and protein folding [12] among others [13].

The availability of quantum hardware allows for benchmarking performance relative to both quantum and classical metrics of computational power. Understanding

benchmarks requires a detailed consideration of how the program and hardware interact as well as how the benchmark metrics represent performance. For example, it is known that performance of the AQO algorithm depends strongly on the specific programming and hardware operation schedules as well as the problem input [14, 15]. Indeed, whereas some studies of the AQO algorithm have reported runtimes that scale polynomially in problem size, others have suggested worst-case exponential behavior or trapping in local minima [16]. An essential step in understanding these analyses is to capture the influence that different programming choices have on these observed behaviors [3, 16, 17, 18, 19, 20].

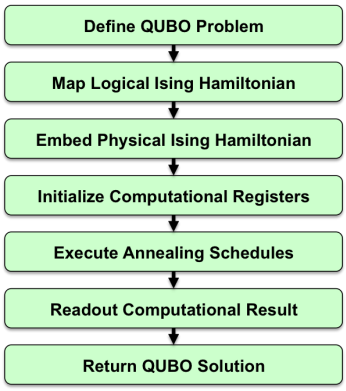

Figure 1. A flowchart highlighting the multiple steps taken to synthesize an adiabatic quantum program for the AQO algorithm. A QUBO problem serves as the classical input while the computed QUBO solution represents the returned result. Each block in the diagram corresponds to an intermediate representation of the program that depends on the choices made in the previous steps.

A significant source of the complexity in analyzing implementations of the AQO algorithm arises from the multiple steps undertaken to synthesize the adiabatic quantum program. The adiabatic quantum programming process illustrated in Fig. 1 begins with the reduction of a classical combinatorial optimization problem to a quadratic unconstrained binary optimization (QUBO) problem that can be mapped into the parameters of an equivalent Ising Hamiltonian. The Ising Hamiltonian must then be mapped onto the processor. This transformation of the reduced problem into a physically realizable program depends on both the hardware layout and the available hardware controls. Ultimately, the computed solution will depend on all previous decisions as well as the actual physics underlying the processor.

It is currently poorly understood how modifications at the various stages in Fig. 1 impact the correctness and efficacy of the computed solutions. Reconciling the seemingly contradictory results from previous studies as well as understanding more recent experimental benchmarks requires investigating how these programming choices impact performance. Motivated by this, we have developed a programming environment that captures each step and synthesizes them together in an integrated workflow. This workflow includes the development of adiabatic quantum programs as well as the collection of diagnostic information for addressing questions about performance. In the absence of actual hardware, we use numerical simulation to evaluate the variety of programming and operational choices that can effect program behavior. With the publication of recent benchmarks from available hardware [19],

we are able to make comparisons between simulated and experimental results.

In this paper we describe the Jade Adiabatic Development Environment (JADE). JADE follows the programming steps highlighted in Fig. 1 and captures both the high-level problem input as well as the low-level quantum physical programming. In addition, we have integrated JADE with a quantum simulation engine that supports user-defined methodologies for running diagnostic analyses. In the current context, we specialize the description of JADE to the case of the AQO algorithm, however, the overall software framework extends to other adiabatic algorithms as well.

This paper is organized as follows. In Sec. 2, we summarize the theoretical background leading to Fig. 1 including the quantum physical basis for AQO. In Sec. 3, we present the model-based design of JADE including the system context, implementations of each component, and our test-driven framework for program verification and validation. We present usage results for the case of a recent benchmark problem in Sec. 4 and we offer conclusions in Sec. 5.

2. Adiabatic Quantum Programming

In this section, we provide a summary of the physical theory and computer science underlying adiabatic quantum programming. This includes the quantum physical description of AQC as well as the steps taken to map the AQO to a hardware control schedule.

2.1. Quantum Computational Model

The physical basis for the AQC model was first established in terms of quantum annealing by Kadowaki and Nishimori [21]. Farhi et al. later formalized these ideas as a means of solving a discrete optimization problem [1]. Several efforts have since shown the equivalence between the AQC model and other quantum computing models [22, 23]. In a generalized AQC algorithm [3], a quantum physical system of n qubits is evolved under the Schrödinger equation

$$i\hbar \frac{\partial \left|\psi(t)\right\rangle}{\partial t} = H(t) \left|\psi(t)\right\rangle \tag{1}$$

according to a time-dependent Hamiltonian

$$H(t) = A(t)H_I + B(t)H_P \tag{2}$$

that interpolates between the initial and final (problem) Hamiltonians, H_I and H_P, respectively, from an initial time $t = t_0$ to a final time $t = T$. We shall assume $t_0 = 0$. In Eq. (2), the schedules $A(t)$ and $B(t)$ satisfy the boundary conditions $A(0) \gg B(0)$ and $A(T) \ll B(T)$, while the quantum system is initially prepared in the lowest-energy eigenstate of H_I. Given the instantaneous eigenvalue equations

$$H(t) \left|\tilde{\phi}_j(t)\right\rangle = E_j(t) \left|\tilde{\phi}_j(t)\right\rangle, \tag{3}$$

with $j = 0, 1, \ldots 2^n - 1$ labeling states of monotonically increasing energy, the initial state condition implies $\left|\psi(0)\right\rangle = \left|\tilde{\phi}_0(0)\right\rangle$.

If the energy gap between the ground and excited states

$$\Delta(t) = \min_{j \neq 0} E_j(t) - E_0(t) \tag{4}$$

is always strictly greater than zero, i.e., $\forall t : \Delta(t) > 0$, then the state $|\psi(t)\rangle$ may remain in the instantaneous ground state with high probability provided certain bounds on the rate of change of the Hamiltonian are satisfied [1]. Consequently, evolution under Eq. (1) to the time T prepares the final state $|\psi(T)\rangle$ in the lowest energy eigenstate of H_P. By making a judicious choice of the final Hamiltonian H_P, the prepared final state may encode the solution to a computation. In order to ensure the computation is correct, the adiabatic condition must be satisfied. This implies that the final time T is chosen to be much larger than the inverse of the minimum spectral gap of $H(t)$ [1]. Failure to ensure this condition risks the possibility that the final state will not belong to the ground state manifold of H_P but rather to an excited state. It is notable that the spectral gap depends not only on the problem to be solved, but also on how the problem is implemented as a quantum program. Understanding input influence on a program run time and error rates is an open question in quantum computer science.

2.2. Adiabatic Quantum Optimization

In specializing to the AQO algorithm, we require a quantum logical system of n qubits with an initial Hamiltonian

$$H_I = -\sum_{i \in V_P} X_i \tag{5}$$

and final Ising Hamiltonian

$$H_P = -\sum_{i \in V_P} \alpha_i Z_i - \sum_{(i,j) \in E_P} \beta_{i,j} Z_i Z_j, \tag{6}$$

where X_i and Z_i are the Pauli operators for the i-th qubit, α_i is the bias on the i-th qubit, and $\beta_{i,j}$ is the coupling between qubits i and j. The graph $G_P = (V_P, E_P)$ with vertex set V_P and edge set E_P defines the input problem as described below. The final Hamiltonian is diagonal in the basis defined by the tensor products of the ± 1 eigenstates of the Z_i operators. This basis will also serve as the computational basis. For comparison, the ground state of the initial Hamiltonian (and initial state of the AQO algorithm) is the symmetric superposition of these computational basis states and has an eigenvalue $-n$.

2.3. Quadratic Unconstrained Binary Optimization

Any binary optimization problem (BOP) can be mapped into the form of the final Hamiltonian in Eq. (6). In doing so, we define the classical input to the AQO algorithm as a quadratic unconstrained binary optimization (QUBO) problem. This is because non-binary as well as constrained optimization problems can be reduced to QUBO [24], and because QUBO has a natural correspondence with the Ising model. The QUBO problem is to find

$$\arg \min_{\mathbf{x} \in \mathbf{B}^m} \mathbf{x}^{\mathbf{T}} \mathbf{P} \mathbf{x}, \tag{7}$$

where $\mathbf{x}$ is a vector of m binary variables with $x_i \in \{0, 1\}$ and $\mathbf{P}$ is an m-by-m symmetric real-valued cost matrix. We interpret $\mathbf{P}$ as a weighted version of the adjacency matrix for the input (problem) graph G_P, introduced in Eq. (6), with $m \equiv |V_P|$ and $(i, j) \in E_P$ iff $P_{i,j} \neq 0$. From this point of view, programming the AQO algorithm requires mapping the matrix $\mathbf{P}$ to the biases and couplings of the Ising Hamiltonian. It has been shown previously by Choi that parameterization of the

logical Ising Hamiltonian in Eq. (6) may be given in terms of the QUBO problem as [14]

$$\alpha_i = \frac{1}{2}P_{i,i} + \frac{1}{4}\sum_{j=1}^{m} P_{i,j} \qquad \text{for } i = 1 \text{ to } m, \tag{8}$$

and

$$\beta_{i,j} = \frac{1}{4}P_{i,j} \qquad \text{for } i < j = 1 \text{ to } m. \tag{9}$$

We may also add an energy shift to the Ising Hamiltonian in Eq. (6) of the form

$$\gamma = \frac{1}{4}\sum_{i,j=1}^{m} P_{i,j} + \frac{1}{2}\sum_{i=1}^{m} P_{i,i} \tag{10}$$

in order to match the energies of the solution state. Although this shift does not affect the solution obtained using AQC, it must be accounted for in reporting the minimal value in Eq. (7).

2.4. Hardware Embedding

Whether or not the logical Hamiltonian in Eq. (6) is supported directly on a given hardware depends on the available connectivity of that hardware. We express the connectivity of a targeted processor in terms of its hardware graph $G_H = (V_H, E_H)$. When any vertex can be coupled to any other vertex and $|V_H| \geq |V_P|$, then it is possible to support all possible input problems using a one-to-one mapping between the logical and physical qubits and the biases and couplings of the physical Hamiltonian. However, when G_H is less than fully connected, then there are certain input problems that will not map directly into hardware. In such circumstances, it may be possible to embed the problem graph G_P into the hardware graph G_H via graph *minor embedding* [25, 15].

We formally define the *minor embedding* of a graph G_P into a graph G_H as a mapping $\phi : V_P \to V_H$ such that:

(i) each vertex i in V_P is mapped to the vertex set of a connected subtree T_i of G_H.

(ii) if $(i, j) \in E_P$, then there exist $\tau_i, \tau_j \in V_H$ such that $\tau_i \in T_i$, $\tau_j \in T_j$, and $(\tau_i, \tau_j) \in E_H$.

If such a mapping ϕ exists, then G_P is *minor-embeddable* in G_H, or G_P is a *minor* of G_H. In subsequent discussions, we simply use the term embedding as a reference to minor embedding.

In adiabatic quantum programming, the vertices of the input graph G_p represent the bits of a candidate solution to the QUBO problem, while the edges represent the presence of nonzero coupling coefficients, as defined in Eqs. (7) and (6) respectively. The vertices of the hardware graph G_H represent the physical qubits and the edges represent the couplings between qubits that are available in the hardware. An embedding maps each vertex in V_P to a subset of V_H and each edge in E_P to edges between these subsets. When an embedding exists, then the resulting subgraph $G^* = (V^*, E^*)$ of the hardware graph defines the physical Ising model

$$H_{G^*} = -\sum_{k \in V^*} \alpha_k^* Z_k - \sum_{(k,\ell) \in E^*} \beta_{k,\ell}^* Z_k Z_\ell \tag{11}$$

The bias and coupling coefficients α_k^* and $\beta_{k,\ell}^*$ depend on the selected embedding ϕ per the requirements (i) and (ii) listed above. The physical Ising coefficients are defined as [25]

$$\alpha_k^* = \alpha_i / |T_i| \qquad \text{for each } k \in V_{T_i} \tag{12}$$

and for $k \neq \ell$

$$\beta_{k,\ell}^* = \begin{cases} \beta_{i,j} / \mathit{edges}(T_i, T_j) & \text{for } k \in T_i \text{ and } \ell \in T_j \text{ and } i \neq j \\ J & \text{for } k \in T_i \text{ and } \ell \in T_j \text{ and } i = j \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

where $\mathit{edges}(T_i, T_j)$ is the number of edges between trees T_i and T_j and the constant J is chosen sufficiently large to force the qubits in each tree to be strongly correlated. Setting these coefficients requires knowledge of the matrix $\mathbf{P}$ and the selected embedding implied by G^* [25, 15]. The embedding need not be unique and, consequently, different instances of the Hamiltonian in Eq. (11) may correspond to the same logical problem of Eq. (7).

A key dependency in finding an embedding is the target hardware graph G_H. The hardware graph defines the vertices and connectivity that are available to express the Ising model. An example hardware graph is shown in Fig. 2. Finding those graphs that can be embedded into a fixed hardware graph is an example of subgraph isomorphism, which is known to be NP-Complete in difficulty. For small hardware graphs, it is tractable to calculate the maximal minors of the graph, i.e., the minors of G_H whose subgraphs represent all other graphs contained in G_H [15]. However, this is a brute force approach and therefore does not scale favorably with hardware size. Alternatives include heuristic algorithms that incorporate knowledge of G_H or that limit the types of input problems.

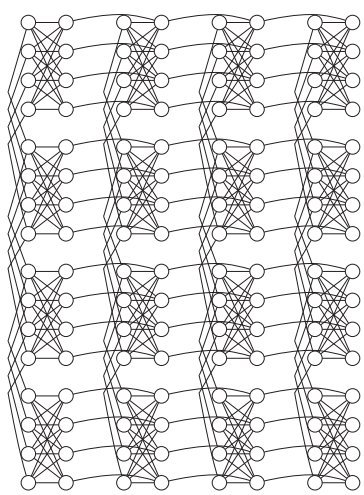

Figure 2. A hardware graph for the *Rainier* processor produced by D-Wave Systems, Inc. The design is a 4×4 lattice of interconnected unit cells, with each unit cell is expressed as a $K_{4,4}$ graph. The geometry of the hardware plays an important role in determining which graphs can be embedded.

2.5. Hardware Schedules and Program Execution

We restrict our discussion to AQC programs that use a time-dependent Hamiltonian fitting the form of Eq. (2), which interpolates between an initial Hamiltonian H_I and the problem Hamiltonian H_P according to the time-dependent annealing schedules $A(t)$ and $B(t)$. More generally, individual biases and couplings can be time-dependent, e.g., $\alpha_i = \alpha_i(t)$. In either case, the time-dependent schedules specify the rate at which the total Hamiltonian $H(t)$ changes and, consequently, they play an important role in the computational error rates. In particular, the final time T needs to be sufficiently large to ensure the validity of the adiabatic condition, namely,

$$T \gg \frac{\mathcal{E}}{\Delta^*} \tag{14}$$

where $\Delta^* = \min_t \Delta(t)$ is the global minimum of the spectral gap defined in Eq. (4) and $\mathcal{E} = \max_t \langle dH(t)/dt \rangle$ is the maximal rate of change during evolution [1]. In the absence of information about $\Delta(t)$, it is difficult to ensure the adiabatic condition is satisifed. This uncertainty is one source of the difficulty in benchmarking adiabatic quantum programs. Recent results on amplifying spectral gaps [26] and developing fault tolerant programs [27] suggest new methods for mitigating this uncertainty.

Although the annealing schedules are sufficient for coarsely specifying program execution, it is ultimately necessary to provide the physical implementation of those schedules in terms of hardware controls. The hardware controls that are available for tuning the biases and coupling of a processor must be capable of expressing programmed schedules. However, available controls are highly dependent on the physics underlying a processor and ensuring the exact implementation of an arbitrary annealing schedule may not be possible. Limitations on annealing schedules arising from constraints and dependencies of control values creates additional uncertainty in the benchmarking effort. Accounting for control constraints and quantification noise is necessary to provide a clear picture of how processor differences impact program behavior. For example, in the case of the family of processors from D-Wave Systems, Inc., biases and couplings can be mapped directly to models for the underlying superconductor Josephson-junction. However, the precision of this mapping is limited by the resolution of the on-board digitial to analog converters (DAC's) [4, 28].

In addition to the constraints expected from hardware design, it is also necessary to anticipate the influence of noise on program behavior. Two types of noise affecting quantum dynamics are classical noise in the controls and quantum noise in the system dynamics. Quantum noise may be modeled as an undesired interaction between computational qubits and non-control elements of the hardware. A specific example is the case of thermal influences on the quantum dynamics, which invalidate the pure state description in Sec. 2 and undermine the adiabatic conditions [29]. Similarly, classical noise in the hardware controls yields a mixed-state description of the quantum dynamics and may bias program execution away from the solution of interest.

Once the time-dependent behavior of the Hamiltonian $H(t)$ has been fully specified, it remains to execute the program. As noted before, the typical sequence begins by initializing the quantum computational register in the ground state of the initial Hamiltonian H_I. How initialization is implemented varies with processor and, more important, it may not be implemented perfectly. This additional source of noise must also be accounted for in evaluating program behavior as it is likely to influence the computational result. The remaining step in execution is to carry out the hardware control schedule and, therefore, the programmed computation.

2.6. Computational Readout and Problem Solution

After evolving to the final time T, the state of the computational register is determined using a suitable measurement or readout method. For the case of the AQO algorithm, the ground states at time T are computational eigenstates and, therefore, readout implies a direct measurement in the computational (Z) basis. As with program execution, it is more realistic to describe the readout process in terms of the hardware controls. This description includes capturing any noise or uncertainty in the measurement process.

The bit string generated from computational readout is the result of the quantum annealing process. However, mapping this result back to a solution for the original QUBO problem requires decoding measurements according to the inverse of the embedding map. For those cases where a tree of physical qubits represents a single logical qubit, it is necessary to check the value of all such qubits. In cases where measurement results within a tree disagree, then various strategies can resolve the uncertainty. One simple example is to use a majority vote. After decoding the computational readout, a solution to the original QUBO problem is produced and the program is complete. It may be necessary to repeat the execution of the program, for example, to gather statistics on the readout or solution states, however, the steps performed are similar to those described above.

3. Jade Adiabatic Development Environment

As presented in Sec. 2, programming the AQO algorithm for an arbitrary QUBO is a highly tunable process. In this section, we describe a software-based implementation of the process that provides control over each of the programming steps shown in Fig. 1. We also describe the integration of this environment with a computational engine that uses numerical simulation for profiling these programs. The simulator is intended for providing insights into how program choices impact program performance.

The Jade Adiabatic Development Environment (JADE) is motivated by the need to provide theoretical benchmarks for current and future adiabatic quantum computing devices. In particular, it was designed to capture insights into the behavior of processor architectures. This is accomplished by using a numerical simulator backend to calculate the time-dependent processor state with respect to programmed algorithm. JADE provides both an engine for simulating the programs that run on adiabatic quantum computing devices and a development environment for specifying program input. In addition, JADE provides methods for constructing adiabatic quantum processor configurations, i.e., the quantum hardware, and for debugging the implementation.

JADE is built using model-driven development, a software development methodology with a strong focus on system use cases as well as architectural extensibility and stability [30]. This methodology allows developers to manage system complexity and rigorously verify and validate the final product implementation. Our model-based approach uses the Unified Modeling Language (UML) to capture design decisions and trace requirements [31]. We also rely heavily on an object-oriented programming paradigm and software design best practices, such as test driven development [32].

3.1. Use Cases

JADE is designed to provide infrastructure for developing AQC programs and a computational engine for simulating them. This includes functionality for parsing input optimization problems, configuring new quantum hardware, and performing program profiling. Given this broad scope in functionality, JADE was designed for two distinct actors: the *Analyst* and the *Engineer*.

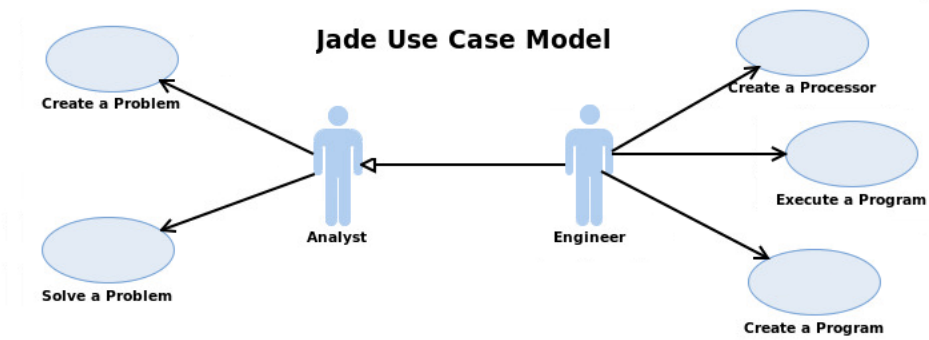

Figure 3. The *Analyst* and *Engineer* actors are distinguished by how they use JADE. The *Analyst* is exposed to only a high-level input problem and its computed solution. The *Engineer* has the ability to tune the low-level programming steps and to analyze the computational readout.

An *Analyst* represents a JADE user whose primary goal is to solve a discrete optimization problem. The *Analyst* requires a development environment that automates programming choices and execution sequences. In contrast, an *Engineer* expects to perform additional programming tasks such as customizing low-level Hamiltonian parameters, constructing specialized processor configurations, and defining embedding maps or annealing schedules. As seen in Fig. 3, this desired JADE functionality is encapsulated by the following use case model:

- *Create a Problem* - the *Analyst* constructs a discrete optimization problem as either a BOP or QUBO problem. In the case of the former, JADE converts the BOP to its corresponding QUBO representation. This use case creates a *Problem* entity.
- *Solve a Problem* - the *Analyst* selects a previously created *Problem* to solve using AQO. This use case returns a *Solution* entity, which is the computed solution to the input problem.
- *Create a Processor* - the *Engineer* creates a processor configuration by specifying the number and connectivity of physical qubits. The *Engineer* may also customize the processor by specifying classical and quantum noise models as well as hardware control constraints. This use case creates a *Processor* entity.
- *Create a Program* - the *Engineer* creates a quantum program that is either a logical program or a physical program. A logical program is synthesized from selected *Problem*, *Processor*, and *Embedding* entities, while a physical program is synthesized only from a *Processor*. For the physical program, the *Engineer*

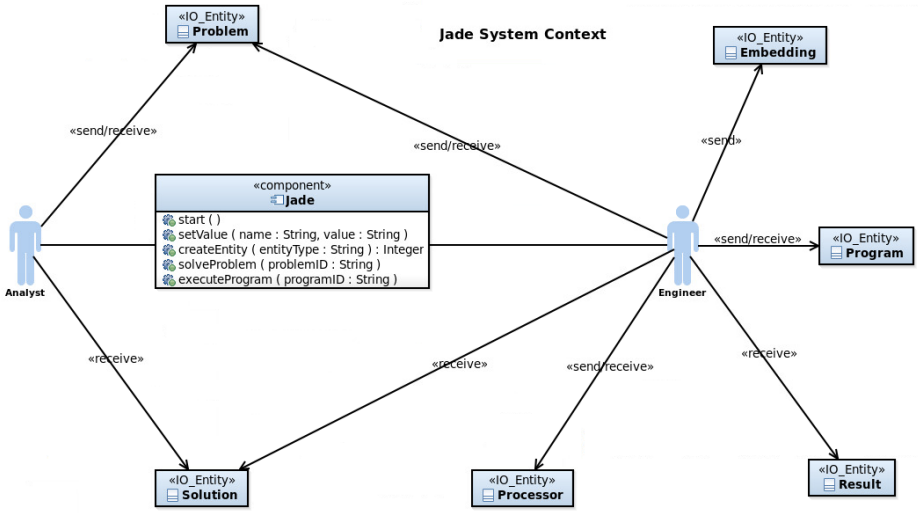

Figure 4. The JADE system context represents the interactions between the *Analyst* and *Engineer* actors with the top-level data entities and the software.

sets the parameters of the final Ising Hamiltonian including biases, coupling, and annealing schedules. Both instances of this use case create a *Program* entity.

- *Execute a Program* - the *Engineer* executes a *Program*. With JADE, the *Engineer* submits the *Program* for simulation along with any profiling and simulations options. This use case creates a *Result* entity that corresponds to the computational readout following program execution.

3.2. System Context

Alongside the use case model, we also present the system context model in Fig. 4. The system context describes the communication between JADE and its environment as driven by the use case model. The system context details how the *Analyst* and *Engineer* interact with the various input-output (I/O) data. As shown in Fig. 4, the six types of I/O data are: *Problem*, *Processor*, *Embedding*, *Program*, *Result* and *Solution*. These I/O entities are further specified in Sec. 3.3.

An *Analyst* only has access to *Problem* and *Solution* entities. However, we anticipate that JADE must synthesize other entities internally, for example, a *Program* is required to generate a *Solution*. Consequently, JADE will need private non-interactive methods for internal synthesis of the remaining entities. Although *Processor*, *Embedding*, and *Program* are generated by the system during the *Analyst* workflow, we do not explicitly model that dependency in Fig. 4.

3.3. Component Architecture

JADE comprises three distinct components: *JadeD*, *Sapphire*, and *NiCE*. The *JadeD* component is responsible for data creation, management, synthesis, and verification, i.e., domain logic. The *Sapphire* component is responsible for the simulation of

quantum programs according to user-defined plug-ins. The *NiCE* component, a pre-existing open source project, is used to integrate the *JadeD* and *Sapphire* components and to manage the computational work flow [33]. Each component provides an independent API.

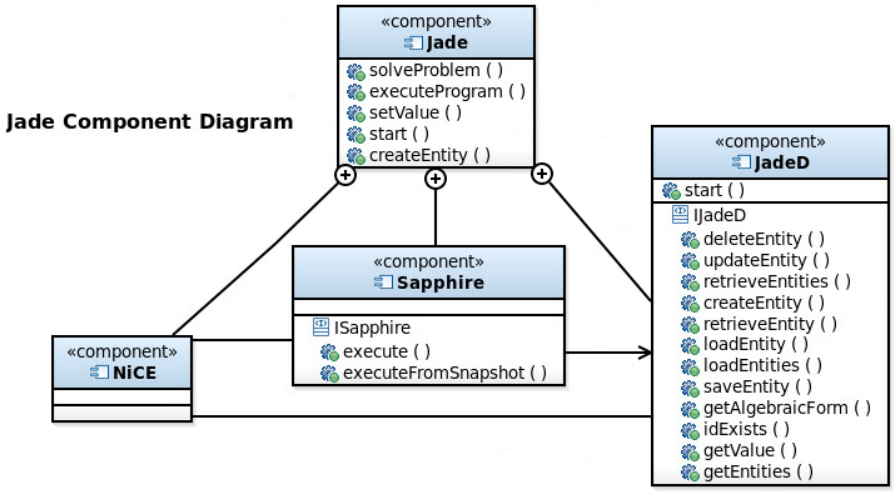

Figure 5. JADE comprises three components: *JadeD*, *NiCE*, and *Sapphire*. The interfaces presented for each component are used to manage component interactions and maintain the separation of concerns between domain logic (*JadeD*), workflow management (*NiCE*), and numerical simulation (*Sapphire*).

Figure 5 highlights the interactions between the three components and the associated interfaces. Both *JadeD* and *Sapphire* couple to *NiCE*, which provides a user-driven coupling between program development and program execution. There is a dependency between *JadeD* and *Sapphire* due the latter's need to parse *Program* data structures. This dependency is restricted to a very narrow subset of the *JadeD* functionality and we expect future versions will isolate it in a separate shared library.

3.4. JadeD

The *JadeD* component handles creation and manipulation of quantum programming by exposing a basic create, retrieve, update, and delete interface. This interface enables generation, manipulation, and persistence of *Entity* data objects, which represent high-level abstractions of the various types of I/O data. The functional scope of *JadeD* includes parsing user-provided input into verified formats, validating that input, and generating subclasses of *Entity* tailored to specific input types. We define an *IJadeD* interface to specify how the *JadeD* component interacts with clients. By defining a formal interface, we are able to offer the option of supporting multiple *JadeD* variants.

As shown in Fig. 6, the *IJadeD* interface includes a number of methods for creating and storing entity instances. The *JadeD* class is a realization of this interface that provides a concrete implementation of the defined functionality. The *JadeD*

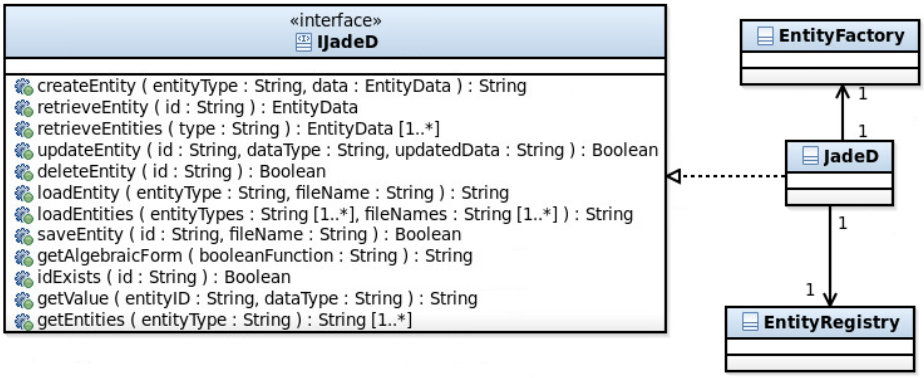

Figure 6. The *IJadeD* interface defines the methods exposed to the user or external application. *JadeD* implements this interface by making use of various data entities that can be generated using a factory pattern and managed using a common registry.

implementation presented here uses a variety of object-oriented design patterns with the factory design pattern being the most significant. The factory pattern is used to create and modify entities in an abstract manner, which pushes the underlying details of construction to the varying entity subclasses. A registry enhances this factory pattern by permitting the sharing of objects across domain boundaries. The use of factories and a data registry allows future developers to add new entity specializations in an easy and efficient manner. In Fig. 6, the factory pattern and the corresponding data registry are implemented as *EntityFactory* and *EntityRegistry*, respectively.

3.4.1. Graph The *graph* data structure represents a set of vertices together with a set of edges coupling those vertices. Graph structures are common to the *Problem*, *Processor*, *Embedding*, and *Program* entities. The *JadeD Graph* model shown in Fig. 7 provides an abstraction of this structure in a way that promotes customization and extensibility with respect to a given entity type.

In supporting this versatility, the *Graph* class utilizes two factory design patterns for generating vertices and edges [30]. This ensures object polymorphism by allowing custom subclasses to inject specialized edges and vertices. For example, this mechanism allows the production of static graphs for *Problem*, graphs that evolve in time for *Program*, and graphs that alter their state according to predefined conditions or controls for *Processor*.

3.4.2. Problem The *Problem* class is a subclass of *Entity* that encapsulates the input data describing a discrete optimization problem. It is created by either an *Analyst* or *Engineer* in order to define the logical problem that the system will solve.

The current implementation of *JadeD* permits users to construct two distinct types of *Problem*. The first is a weighted or pseudo-Boolean optimization problem. The user inputs an arbitrary number of Boolean clauses in terms of the literals b_i, e.g., $((b_1 \text{ AND } b_2) \text{ OR NOT } b_3)$, and each clause also has an associated real-valued weight w_i. The pseudo-Boolean function is then cast into an equivalent BOP by

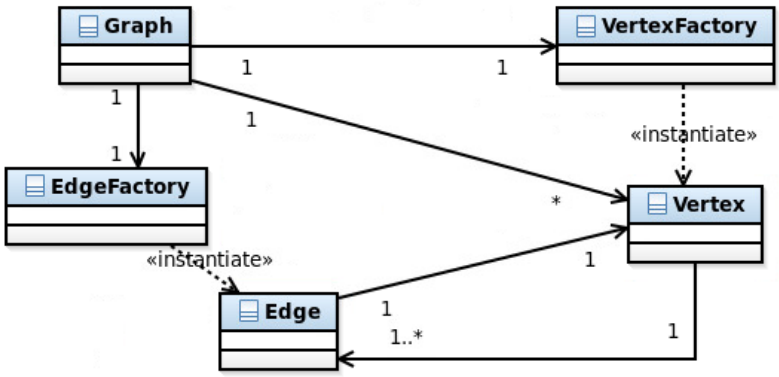

Figure 7. The *Graph* class encapsulates vertices and edges, whose respective implementations use the *VertexFactory* and *EdgeFactory* factory patterns.

converting each Boolean literal to a corresponding binary variable, e.g., $b_i \mapsto x_i$, $\textsc{True} \mapsto 1$ and $\textsc{False} \mapsto 0$. The Boolean clauses are then recast into equivalent binary arithmetic expressions. Denoting the i-th binary arithmetic clause as f_i and the corresponding weight as w_i, the equivalent BOP over n bits is

$$\arg\min_{\mathbf{x}} \sum_i w_i f_i(\mathbf{x}), \tag{15}$$

where $\mathbf{x} \in \{0,1\}^n$ is an n-bit vector [24]. In JADE, the *BOP* class stores both the original Boolean clauses and the reductions to algebraic expressions with corresponding weights.

The second type of *Problem* supported by *JadeD* is the QUBO problem defined in Eq. (7). For this type, the input corresponds to the elements of the matrix $\mathbf{P}$. The matrix $\mathbf{P}$ is then interpreted as a weighted adjacency matrix and parsed by *JadeD* into a *Graph*. Accordingly, the *QUBO* class is a subclass of *Graph*. The dependencies between the various *Problem* subclasses are illustrated in Fig. 8.

As discussed in Sec. 2, a BOP of the form in Eq. (15) can be reduced to a corresponding QUBO problem of the form in Eq. (7). The reduction, however, requires introduction of penalty terms to replace multilinear terms with quadratic or linear terms [24]. Expressing these penalties ultimately requires additional ancilla bits which enlarge the binary state space. When *JadeD* instantiates a *BOP*, the corresponding *QUBO* is immediately generated as part of the *Problem*. The relevant *BOP* information is maintained as part of the *Problem* in order to facilitate developing the *Solution* entity returned to the *Analyst*.

3.4.3. Processor The *Processor* entity encapsulates the structure and behavior of a quantum hardware configuration. It generalizes *Graph* by using an adjacency matrix with unit diagonal entries to indicate vertex availability and unit off-diagonal entities for available connections between qubits. *Processor* wraps a subclass of *Graph* referred to as *Hardware* and provides methods to query and manipulate its structure. The *Hardware* subclass can also implement the embedding of an input *Problem* into the

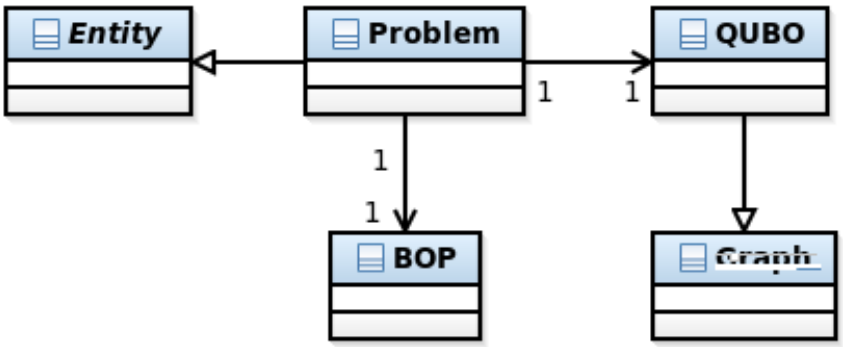

Figure 8. The dependencies of *Problem* on *BOP* and *QUBO* entities. *Problem* generates *QUBO* from an input *BOP*. Alternatively, the *QUBO* may be supplied directly.

hardware. This produces an *Embedding* entity, which subclasses *Graph* to express the graph G^* that defines the embedded H_{G^*} from Eq. (11).

Processor also allows users to specify a functional time dependence for the bias and coupling parameters of vertices. The *Control* class encapsulates a set of functions that give users the ability to manipulate physical quantities that directly influence the physics of the quantum hardware. Custom noise models can also be added to *Processor* through the *Noise* class, which can express both classical and quantum noise functions.

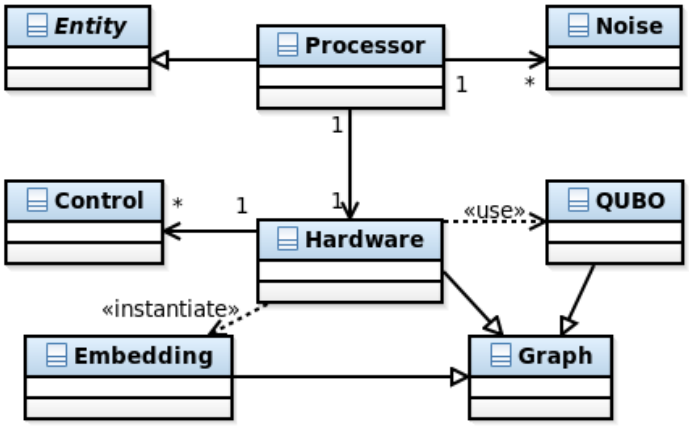

Figure 9. The dependencies of the *Processor* class, which includes *Hardware*, *Noise*, and *Control* entities. The *Embedding* entity is instantiated after a QUBO is embedded into the *Processor*.

3.4.4. Program The *Program* class is a subclass of *Entity* that is used to synthesize specific instances of *Problem* and *Processor* into an implementation of the adiabatic quantum optimization algorithm. A *Program* is the primary input to the *Sapphire* simulation component and two different types can be constructed, physical or logical. The main difference between these two types for *Program* is the presence or absence of a high-level logical *Problem* definition.

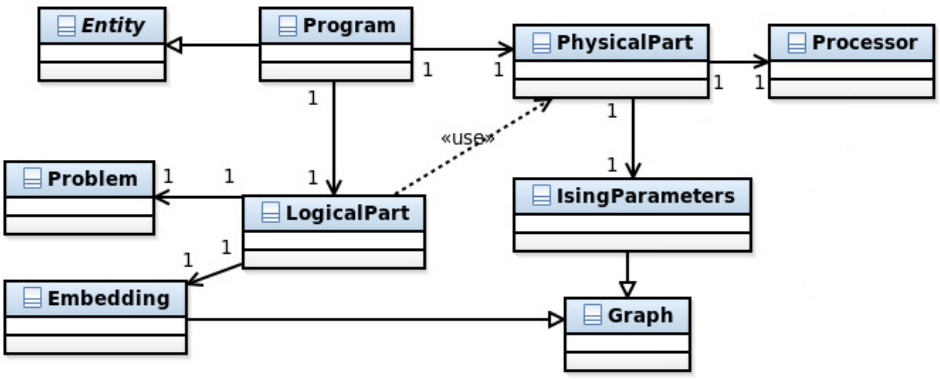

Figure 10. The dependencies of the *Program* class. The presence of a *Problem* distinguishes a logical program from a physical program, while both class types have an associated *Processor*.

As shown in Fig. 10, type-switching is accomplished by composing *Program* with two classes: *Logical Part* and *Physical Part*. The physical part of a *Program* encapsulates the physical representation of the time-dependent Hamiltonian defined in Eq. (2). This includes a reference to a *Processor* and the parameters defining the final Ising Hamiltonian as well as the annealing schedule for each qubit. The logical part of a *Program* encapsulates a physical program as well as a reference to the specified *Problem* entity that is being solved. While the physical part of a *Program* entity is always required, the logical part it is not. For *Analyst* use cases, the *Program* always has a logical part. In the absence of a logical input, the *Program* corresponds to an *Engineer* defined instance of an Ising Hamiltonian.

The mapping of the *Logical Part* into the *Physical Part* generates an *Embedding* of the *Problem* into the *Processor*. As described in Sec. 2, embedding generates a map between each logical vertex and a subgraph in the *Processor*. Within *JadeD*, this is accomplished using a subclass of *Graph* called *Embedding*. The *Embedding* class finds an embedding of the *Logical Part* into the provided *Processor* and *Hardware*. The current *Embedding* class supports the maximal minors methods described by Klymko et al. [15]. Its use is limited to a $K_{4,4}$, but the extensibility of *Embedding* means that the additional, greedy methods described by Klymko et al. can also be incorporated.

3.5. NiCE

The *NiCE* component is responsible for accepting user input, returning JADE output, and managing the computational workflow. It also provides a graphical frontend for JADE. *NiCE* is an existing open-source project that was leveraged for reducing development time and ensuring extensibilty. In addition to I/O management, the

NiCE component orchestrates the interactions between the *JadeD* and *Sapphire* components. It enables users to create input files, launch simulations and examine program metrics.

NiCE is based on a client-server model, where the server handles primary data management and the client acts as the user frontend. It is also possible for the server to manage remote workloads including, for example, simulations launched on remote hosts. We use the *NiCE* server as the primary means for launching and monitoring numerical simulations on both local and remote machines.

We have developed several plug-ins for *NiCE* that allow direct interaction with the *JadeD* component for the creation and revision of the *Problem*, *Processor*, and *Program* entities. A screenshot of one such *NiCE* form is provided in Fig. 11. *NiCE* is based on the Open Source Gateway Initiative (OSGI) framework that, among other things, permits dynamic registration of services. We use *NiCE*'s implementation of dynamic registration to recognize and load user-defined plug-ins into JADE. This feature permits, for example, user-defined methods for simulation that are developed independently from JADE to be added during runtime. Additional information about *NiCE* is available from its website [33].

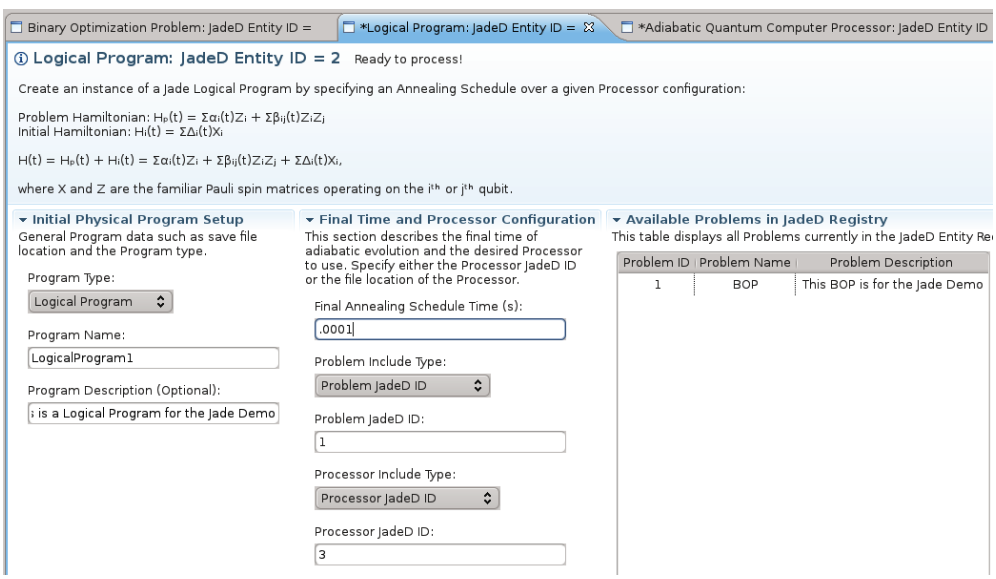

Figure 11. A cropped screenshot from the *NiCE* client for JADE showing the synthesis of a *Program* from a logical *Problem* and a selected *Processor*.

3.6. Sapphire

Sapphire is the JADE component responsible for profiling *Program* entities. This includes carrying out numerical simulations of the quantum dynamics as well as other characterizations such as computing the time-dependent energy eigenspectra and computational error rates. While its primary use is to compute the *Result* of a *Program*, *Sapphire* permits a robust set of possible use cases. This is a result of our use of a plug-in architecure to support user-defined extensions to *Sapphire*.

For example, numerical simulation techniques can be tailored to specific questions or physical assumptions. This promotes analysis at any desired fidelity and gives the user the ability to compare different simulation techniques against experimental benchmarks.

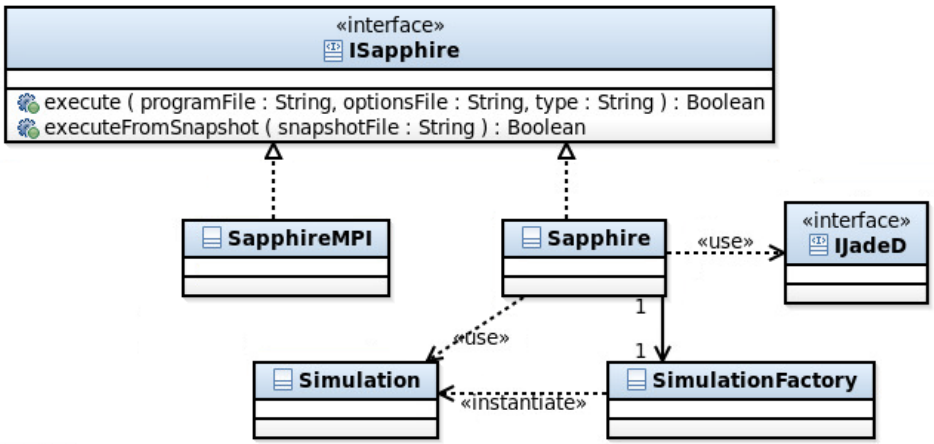

Figure 12. The *ISapphire* interface expresses both the *Sapphire* and *SapphireMPI* classes. *Sapphire* makes use of the factory design pattern for generating *Simulations* that are labeled by the *type* argument to execute. *SapphireMPI* has an identical structure.

The extensibility of *Sapphire* is achieved through the interplay of a number abstractions and design patterns, as shown in Fig. 12. *Sapphire* only exposes a few methods to external clients through the *ISapphire* interface. This decoupling between behavioral definition and actual implementation allows *Sapphire* to take on a number of varied forms. For example, JADE currently provides a *Sapphire* implementation for multi-threaded, shared memory architecture. We have also implemented *SapphireMPI*, which uses the MPI (Message Passing Interface) library to execute simulations on distributed architectures. The most significant difference between the two implementations is the MPI dependency and the need to perform unique initialization steps for *SapphireMPI* prior to beginning the numerical simulation.

All implementations of *Sapphire* must define the method *execute*. When *execute* is invoked, *Sapphire* utilizes the *JadeD* file-parsing capabilities to construct the *Program* object defining the parameters of the numerical simulation. *Sapphire* next parses the simulations options provided by the user to create a *Simulation* object using the *SimulationFactory*. The *Simulation* class is the basis for the extensibility of *Sapphire* using plug-in libraries. A plug-in is essentially a subclass of *Simulation* that provides a specialized numerical or algorithmic approach to simulation.

3.7. Simulation Plug-ins

The *Simulation* class is the primary unit of functionality within *Sapphire* and it is used to encapsulate a specific mathematical evolution of a quantum state. The factory design pattern allows *Sapphire* to remain completely agnostic to simulation details.

However, there is a specific sequence of execution statements that are part of *Sapphire*. Program execution always begins with an initialization statement followed by a loop over a time-dependent solver. Once the exit condition is met, i.e., when $t = T$, the computational state undergoes readout before the program issues finalization commands. All plug-ins for *Sapphire* must adhere to the *Simulation* class functionality defined below.

- *initialize*: This method is used primarily to initialize quantum state of the simulation. Additional task include setting up any pre-simulation conditions or parameters.
- *anneal*: This method is called every time step by *Sapphire* to advance the system's quantum state. Developers should implement this method to update the state vector with the mathematics inherent to their specific technique for solving the time-dependent Schrödinger equation.
- *queryState*: This method is used to query the state of the simulation, including the computational state of the simulated program. The output generated by this method is highly variable and it can include the internal representation of the quantum computational state or the complete eigenenergy spectrum can be written to an output file. These output files can then be used as checkpoints for restarting the simulation.
- *measure*: This method is called after *anneal* completes and it represents measurement of the final computational state.
- *finalize*: This method is used for any final calculations or clean up routines.

Developers of simulation plug-ins must subclass *Simulation* and implement the purely virtual *anneal* method. All other methods have default implementations that can be overwritten for specialized functionality. JADE also provides a specialized *HamiltonianGenerator* abstraction that permits decoupling of numerical dynamics from the actual form of the Hamiltonian describing the system.

3.7.1. Plug-in Examples The *Sapphire* plug-in architecture maintains extensibility to new simulation methodologies. A plug-in represents a user-created library that implements the *Simulation* class defined above. JADE users are therefore able to tailor quantum computing simulation techniques to specific problems or metrics of interest. We provide examples of plug-ins that implement *Simulation* below.

- *SimulationZero*: This plug-in provides a zero-th order approximation about the state of the computational register. Specifically, this simulation calculates the time-dependent eigenspectrum and instantaneous eigenstates of the time-dependent Hamiltonian defined by a *Program*. SimulationZero does not provide information about the quantum dynamics but essentially diagonalizes the Hamiltonian at each time step. This analysis provides information about the time-dependent energy gap. Our implementation makes use of the Eigen library, which is an open-source C++ template library for linear algebra [34].
- *RK4Simulation*: This plug-in provides a fourth-order Runge-Kutta solver for the time-dependent Schrödinger equation as in Eq. (1). RK4Simulation uses two time steps, one for the outer *anneal* method which updates the Hamiltonian and a second for the inner *evolve* loop that numerically solves a finite-difference equation. For each *evolve* time step, the plug-in updates the quantum state

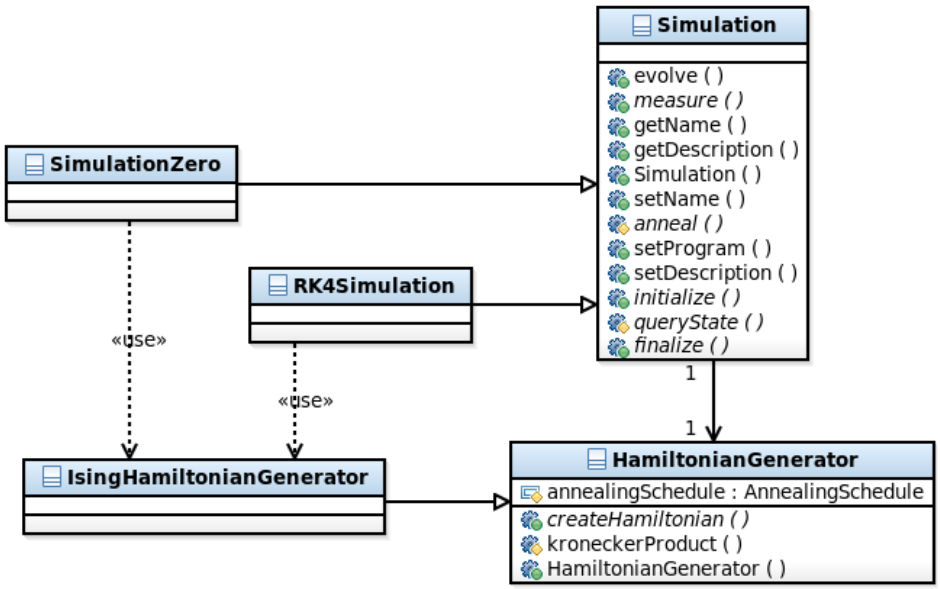

Figure 13. Examples of the plug-ins that implement the *Simulation* class. SimulationZero and RK4Simulation both implement subclass *Simulation* and also make use of IsingHamiltonianGenerator, a subclass of the *HamiltonianGenerator*.

and for each *anneal* it computes the instantaneous eigenspectrum. The plug-in also implements the *queryState* method to provide a *Snapshot* output that contains details about the computational state and eigenspectrum. Simulation options include the time steps, number of *Snapshot* files created, and number of eigenstates reported by *queryState*. This plug-in also makes use of the linear algebra functionality provided by the Eigen library.

- *FOPSimulation*: The FOPSimulation plug-in is based on a first-order perturbative solution to the time-dependent Schrödinger equation. It evolves a pure state according to a first-order Magnus expansion for the time-dependent propagation operator. Numerically, the propagation operator is diagonalized by the *anneal* method and applied successively to the state during the *evolve* method. This method has an error of $\mathcal{O}(\Delta t^3)$. Similar to the other simulation methods, Eigen is used to perform the matrix exponential and matrix-vector multiplications.

3.8. Testing Framework

The design and implementation of JADE relies heavily on test-driven development. A formal and rigorous testing model was defined before any actual product code was developed. This has ensured that (1) the functionality of each test unit was defined prior to its implementation and (2) the implementation of each source unit was fully compliant with the predetermined functionality. We employed test-driven development by modeling and designing surrogate classes whose sole purpose was for

unit testing critical behavior in actual JADE classes. An example is shown in Fig. 14, where we test the *Simulation* class using surrogates for most objects in the Sapphire component. There is a corresponding *SimulationTester* class. Every class in JADE has a corresponding test class in order to provide the greatest assurance that the code adheres to design requirements.

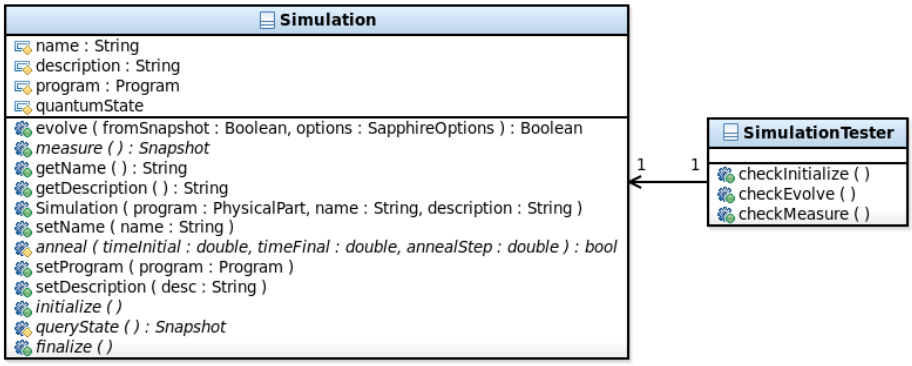

Figure 14. *SimulationTester* is external to *Simulation* but capable of accessing its methods.

4. Benchmark Demonstrations

As an example of how JADE can be used for benchmarking quantum programs, we present results based on the recent experimental benchmarks reported by Boixo et al. [19]. Their work was performed on the Rainier processor from D-Wave Systems, Inc. and used the 8-qubit Ising model represented in Fig. 15.

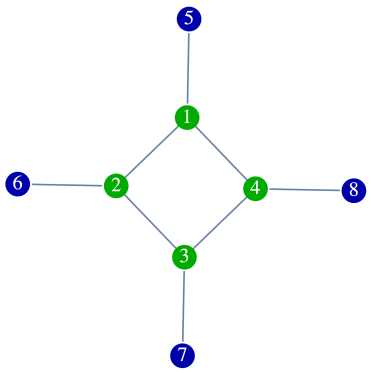

Figure 15. *The graphical representation of the 8-qubit Ising model investigated by Boixo et al.[19]. Vertices 1-4 (green) represent biases of* $+1$ *and vertices 5-8 represent biases of* -1*. All the edges represent* $+1$ *couplings between connected vertices.*

Boixo et al. showed both theoretically and experimentally that the 8-qubit model in Fig. 15 exhibits a unique behavior. This particular 8-qubit problem exhibited a distinctive behavior that differentiates between the quantum and classical annealing dynamics. The Ising Hamiltonian has a 17-fold degenerate ground state. They used multiple runs of the developed program on the Rainier processor to recover all 17 ground states from computational readout.

We have used the benchmark developed by Boixo et al. to demonstrate the functionality of JADE. Specifically, we defined an 8-qubit *Processor* supporting the $K_{4,4,}$ (bipartite) connectivity familiar from the unit cell in the Rainier processor as shown in Fig. 2. We used an *Embedding* entity based on the maximal minor method discussed by Klymko et al. [15] and we matched the mapping taken by Boixo et al. We programmed linear annealing schedules, i.e., $A(t) = t/T$ and $B(t) = 1-t/T$, and a final time of $T = 30$. We neglected constraints on the controls, as the Ising parameters were very simple, and we neglected all forms of noise in the hardware.

The developed *Program* entity was then simulated using the RKSimulation plug-in described in Sec. 3.7.1. The simulation options given to this fourth-order Runge-Kutta finite-difference solver invoked a quasi-static approximation for the Hamiltonian. That is to say, we used an *evolve* time step of 0.0001 time units with updates to the Hamiltonian made during every *anneal* with a time step of 0.05 time units. The computational registers were initialized to the exact ground state of the initial Hamiltonian in Eq. (5). For diagnostics, we computed the complete eigenspectrum every 3 time units and output both the spectrum and the complete quantum state as part of a *Snapshot*. The *measure* method returned an ordered listing of the output states with their associated probabilities in the generated *Result* entity.

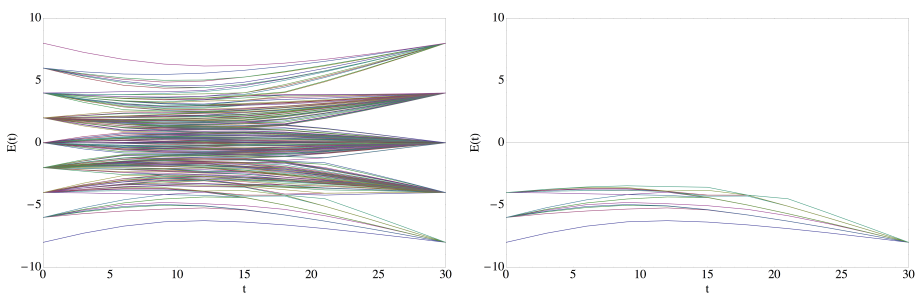

Figure 16. (left) The complete time-dependent eigenspectrum of the 8-qubit benchmark. (right) Time-dependent spectrum for those states terminating in computational ground states. Spectra are computed every 3 time units for a total of 11 points for each of the 256 spectra lines.

The complete time-dependent eigenspectrum computed by JADE is shown in the left panel of Fig. 16. This consists of $2^8 = 256$ lines representing the time-dependent energies of the 256 eigenstates of the Hamiltonian. At the final time T, there are 17 ground states with eigenenergy -8. This matches the eigenenergy and degeneracy derived by Boixo et al. The 17 time-dependent spectra that result in a ground state at the final time are shown in the right panel of Fig. 16. The presence of kinks in the plot indicate that several states undergo avoided crossings with higher energy levels. We also see that the definition of the spectral gap $\Delta(t)$ in Eq. (4) did not distinguish between those instantaneous excited states that terminate in the final ground state manifold from those excited states that remain excited at time T. States terminating

Table 1. Degenerate ground states of the 8-qubit model and their computed probabilities.

Decimal	Binary	Probability
0	0000 0000	0.0582245
1	0000 0001	0.0598409
2	0000 0010	0.0598409
3	0000 0011	0.0620211
4	0000 0100	0.0598409
5	0000 0101	0.0627384
6	0000 0110	0.0620211
7	0000 0111	0.0651488
8	0000 1000	0.0598409
9	0000 1001	0.0620211
10	0000 1010	0.0627384
11	0000 1011	0.0651488
12	0000 1100	0.0620211
13	0000 1101	0.0651488
14	0000 1110	0.0651488
15	0000 1111	0.0677486
255	1111 1111	4.79745×10^{-4}

in the ground state are not computational errors, but transitions from those states to higher lying excited states can contribute to the observed error rate.

The computed populations for the 17 ground states at time T are presented in Table 1 alongside the corresponding computational basis state. It is evident that the first 16 states, i.e., the manifold of states with qubits 1-4 in the 0 (spin down) state, have approximately equal probability while the 17-th state is roughly two orders of magnitude less. However, all the ground states are significantly more likely than the 18-th most probable state, which has a probability much less than 10^{-6}.

The time-dependence of the instantaneous population in the computational basis is shown in Fig. 17. Recall that the system is initialized in the singular computational ground state, as indicated by maximum probability at time $t = 0$. As time progresses, the population remains in the instantaneous ground state until $t \approx 0.9T$ At this point in the program schedule, the energy gap between the ground state and the lowest lying excited states has narrowed sufficiently to permit population transfer, thereby violating the adiabatic condition. Here, the lowest-lying excited states represent instantaneous states that will terminate in the ground state at time $t = T$. There are 16 such states participating in the apparent convergence to approximately 15/16 of the total probability and, as shown in Table 1. The 17-th ground state is not visible in this plot, due to the scale of its contribution, however it undergoes a similar behavior and contains approximately $1/16^2$ of the population. The final 15/256 of probability is distributed over the remaining 239 excited states.

Our simulation of the 8-qubit benchmark appears to be in qualitative agreement with the experimental and theoretical results of Boixo et al. [19]. However, there are several key differences between their program and ours. First, the annealing schedules used by Boixo et al. are not linear and we expect that impacts our comparison

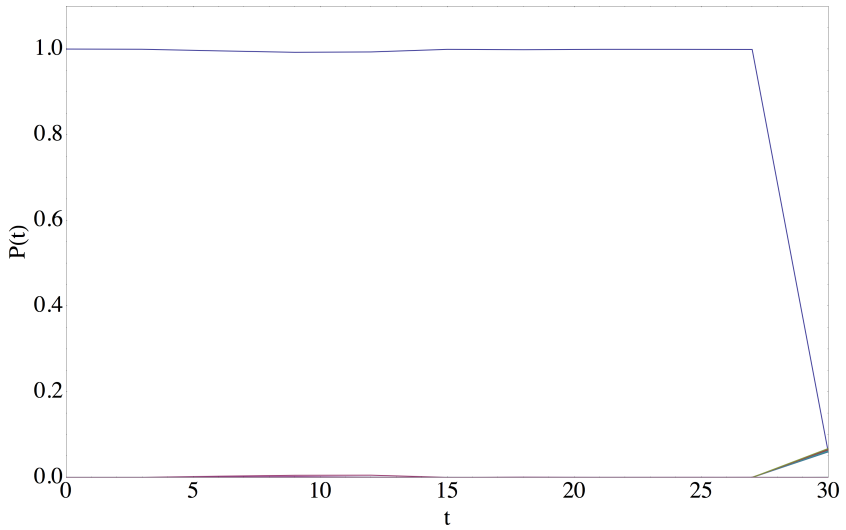

Figure 17. *Time-dependence of the population in the computational basis. The resolution is 11 points for a range of* $T = 30$.

of observed and computed probabilities. Second, we have not incorporated any sources of noise into our simulation studies, whereas previous experiments on the Rainier processor have suggested the influence of thermal noise may be significant. Nevertheless, our intention of this demonstration has been to provide a verifiable example that JADE is useful for benchmark analyses.

5. Discussion

The present availability and continuing development of adiabatic quantum computing hardware opens up new avenues of research for defining methods of quantum programming and computational benchmarking. Benchmark studies are necessary for measuring actual computational power of processors and for improving programming practices. Test vectors appropriate for benchmark studies must be well-defined and the associated difficulty well-understood in order to reliably measure the influence of programming and processor methodologies. We have developed a software environment that offers an interactive approach to adiabatic quantum programming. Most important, JADE parameterizes the programming process and offers opportunities for tuning each step. JADE also provides a plug-in architecture to enable extension to functionality through user-defined programming, simulation, and diagnostic methodologies.

The adiabatic quantum programming sequence summarized in Fig. 1 is sufficient for our current paradigm. However, we do not claim that it is necessary. The current approach is certainly insufficient for other models of quantum computation, such as gate-based models, where fault-tolerant protocols and quantum error correction (QEC) add significant overhead and management complexity. Nevertheless, JADE exemplifies the type of programming environment currently needed by the quantum computer science community.

Acknowledgments

This work was supported by the Lockheed Martin Open Innovation Program at Oak Ridge National Laboratory. The authors thank Greg Tallant and Peter Stanfill of the Lockheed Martin Aeronautics Division for technical assistance. H. S. thanks the Department of Energy Science Undergraduate Laboratory Internship (SULI) program. T. S. H. thanks Owen S. Humble for technical discussions. This manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

References

[1] Farhi E, Goldstone J, Gutmann S, Lapan J, Lundgren A and Preda D 2001 *Science* **292** 472–476
[2] Childs A M, Farhi E and Preskill J 2001 *Phys. Rev. A* **65** 012322
[3] Dickson N G and Amin M H S 2012 *Phys. Rev. A* **85** 032303
[4] Harris R, Johnson M, Lanting T, Berkley A, Johansson J, Bunyk P, Tolkacheva E, Ladizinsky E, Ladizinsky N, Oh T, Cioata F, Perminov I, Spear P, Enderud C, Rich C, Uchaikin S, Thom M, Chapple E, Wang J, Wilson B, Amin M, Dickson N, Karimi K, Macready B, Truncik C, and Rose G 2010 *Phys. Rev. B* **82** 024511–024526
[5] Neven H, Rose G and Macready W G 2008 Image recognition with an adiabatic quantum computer I. Mapping to quadratic unconstrained binary optimization (*Preprint* `0804.4457v1[quant-ph]`)
[6] Neven H, Denchev V S, Rose G and Macready W G 2008 Training a binary classifier with the quantum adiabatic algorithm (*Preprint* `0811.0416[quant-ph]`)
[7] Pudenz K and Lidar D 2012 *Quantum Info. Processing* 1–44
[8] Gaitan F and Clark L 2012 *Phys. Rev. Lett.* **108**(1) 010501
[9] Bian Z, Chudak F, Macready W G, Clark L and Gaitan F 2013 *Phys. Rev. Lett.*
[10] Gaitan F and Clark L 2013 Graph isomorphism and adiabatic quantum computing (*Preprint* `1304.5773[quant-ph]`)
[11] Neigovzen R, Neves J L, Sollacher R and Glaser S J 2009 *Phys. Rev. A* **79**(4) 042321
[12] Perdomo-Ortiz A, Dickson N, Drew-Brook M, Rose G and Aspuru-Guzik A 2012 *Nature Scientific Reports* **2** 571
[13] V N Smelyanskiy et al 2012 A Near-Term Quantum Computing Approach for Hard Computational Problems in Space Exploration (*Preprint* `1204.2821[quant-ph]`)
[14] Choi V 2011 *Quantum Inf. Process* **10** 343–353
[15] Klymko C F, Sullivan B D and Humble T S 2012 Adiabatic quantum programming: Minor embedding with hard faults (*Preprint* `1210.8395v2[quant-ph]`)
[16] Altshuler B, Karvi H and Roland J 2011 *Proc. Natl. Acad. Sci. USA* **108** E19–E20
[17] Farhi E, Goldstone J, Gutmann S and Sipser M 2000 Quantum computation by adiabatic evolution (*Preprint* `0001106[quant-ph]`)
[18] Dickson N G and Amin M H S 2011 *Phys. Rev. Lett.* **106** 050502
[19] Boixo S, Albash T, Spedalieri F M, Chancellor N and Lidar D A 2014 *Nature Comm.* **4** 2067
[20] McGeoch C C and Wang C 2013 Experimental evaluation of an adiabiatic quantum system for combinatorial optimization *Proceedings of the ACM International Conference on Computing Frontiers* CF '13 (ACM) pp 23:1–23:11 ISBN 978-1-4503-2053-5
[21] Kadowaki T and Nishimori H 1998 *Phys. Rev. E* **58** 5355–5363
[22] Aharonov D, van Dam W, Kempe J, Landau Z, Lloyd S and Regev O 2007 *SIAM Journal on Computing* **37** 166–194
[23] Mizel A, Lidar D A and Mitchell M 2007 *Phys. Rev. Lett* **99** 070502
[24] Boros E and Hammer P L 2002 *Dis. App. Math.* **123** 155–225
[25] Choi V 2008 *Quantum Inf. Process* **7** 193–209
[26] Somma R and Boixo S 2013 *SIAM Journal on Computing* **42** 593–610
[27] Lidar D A 2008 *Phys. Rev. Lett.* **100**(16) 160506
[28] M W Johnson et al *Superconductor Sci. Technol.* 065004
[29] N G Dickson et al 2013 *Nature Communications* **4** 1903

[30] Nolan B, Brown B, Balmelli L, Bohn T and Whali U 2008 *Model Driven Systems Development with Rational Products* (IBM)
[31] 1995 *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley)
[32] Bittner K and Spence I 2003 *Use Case Modeling* (Addison-Wesley)
[33] 2013 NiCE Software Project URL `http://niceproject.sourceforge.net`
[34] 2013 Eigen Software Project URL `http://eigen.tuxfamily.org`