AFRL-RI-RS-TR-2014-193

A QUANTUM COMPUTING APPROACH TO MODEL CHECKING FOR ADVANCED MANUFACTURING PROBLEMS

UNIVERSITY OF SOUTHERN CALIFORNIA –
INFORMATION SCIENCES INSTITUTE

JULY 2014

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2014-193 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
STEVEN L. DRAGER
Work Unit Manager

/ S /
MARK H. LINDERMAN
Technical Advisor, Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JULY 2014	FINAL TECHNICAL REPORT	NOV 2012 – JAN 2014

4. TITLE AND SUBTITLE

A QUANTUM COMPUTING APPROACH TO MODEL CHECKING FOR ADVANCED MANUFACTURING PROBLEMS

5a. CONTRACT NUMBER
FA8750-13-2-0035

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Federico M. Spedalieri, John Damoulakis

5d. PROJECT NUMBER
HACM

5e. TASK NUMBER
SL

5f. WORK UNIT NUMBER
QC

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Southern California – Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292-6601

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2014-193

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This project studied the feasibility of integrating the capabilities of the D-Wave adiabatic quantum annealing processor into a Model Checking (MC) approach based on Counter-example Guided Abstraction Refinement (CEGAR). The computational bottleneck of this approach is the solution of certain combinatorial optimization problems for which the D-Wave processor was specifically designed. We developed a set of tools to sidestep the restrictions imposed by the limited connectivity of the processor, performed a set of benchmarking tests of the device, and implemented a proof of concept example that integrated the quantum processor with regular model checking techniques.

15. SUBJECT TERMS

Adiabatic Quantum Annealing Processor, Model Checking, Quadratic Unconstrained Binary Optimization, Integer Linear Programming, Counter-example Guided Abstraction Refinement

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**STEVEN L. DRAGER**
U	U	U	UU	67	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

Section **Page**

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

Table **Page**

1 SUMMARY

The goal of the Quantum Computing Approach to Model Checking for Advanced Manufacturing Problems (QCHECK) research project was to determine if it is feasible in the future to speed up a Model Checking (MC) approach based on Counter-example Guided Abstraction Refinement (CEGAR) by using a D-Wave open system, adiabatic quantum annealing processor. These are specialized computing devices that solve spin Ising models, which are equivalent to Quadratic Unconstrained Binary Optimization (QUBO). We focused on two aspects of the CEGAR approach that involved solving integer linear programs (ILPs) and Boolean satisfiability (SAT) problems. The project was divided in five tasks:

Task1: Study feasibility of using a D-Wave to solve Bounded Model Checking (BMC) problems and implementing Binary Decision Diagrams (BDD) based techniques.
Task 2: Compare the performance of a second generation D-Wave (DW2) on MAX-2-SAT problems native to its architecture, versus the heuristic solver MaxWalkSat.
Task 3: Develop a heuristic embedding algorithm for the DW2 to get around the limited connectivity of the processor.
Task 4: Integrate the CEGAR approach with the DW2 processor.
Task 5: Implement examples.

Results:

Task 1: It was found that even though in principle the required BMC problems could be cast as QUBO problems, the probabilistic nature of the processor (that provides no guarantees that the best possible solution has been found) made the approach susceptible to false negatives: a SAT formula could be proclaimed "unsatisfiable" because the best solution found by DW2 does not satisfy the formula, while a better solution might exist that proves the formula satisfiable. With regards to implementing BDD based approaches using DW2, it was concluded that encoding such a problem as an optimization problem, though possible, would not scale well with system size.

Task 2: The comparison was performed on a set of random instances of MAX-2-SAT that are native to the DW2 processor's architecture, for different numbers of variables. The performance of DW2 was shown to be better than that of MaxWalkSat, with the caveat that MaxWalkSat was not optimized for the DW2 architecture. The issue of DW2's performance vs. that of classical solvers remains (as this report is being written) an open and very contested research topic.

Task 3: A tool to perform heuristic embeddings was created. It allowed us to implement QUBO problems that had a different connectivity graph than the native DW2 architecture. This tool is useful for optimization problems, but suffers the same limitations found in Task 1 for decision problems (i.e., SAT) due to the lack of a guarantee that the solution found is the best possible. The tool needs further optimization, and alternative approaches need to be investigated.

Task 4: Several issues arose that made this task more challenging than was anticipated:

1. Off-the-shelf model checking packages do not provide access to internal data such as the ILP needed to be solved during the CEGAR implementation. We contacted the developers but they were reluctant to give us access to the source code.
2. Many simple models that we studied generated trivial ILPs during the CEGAR implementation, i.e., the solution could be found by simple inspection. It took some time to learn from these models what properties of the system will lead to non trivial ILPs.
3. In order to have access to the ILPs we ended up writing a small model checking package using publicly available libraries. We used the And-Inverter Graph (AIG) format, allowing us to generate small, non-trivial ILPs (on the order of tens of linear constraints and binary variables).

Task 5: We found an example from the literature of a flight control system and checked a safety property that requires two exclusive flight modes not to be engaged at the same time. We implemented this problem integrating the AIG-based model checker with the DW2, ran the CEGAR approach starting with an abstraction that had 33 hidden variables. Our integrated code proved the system to be safe by making visible only 14 of the 33 hidden variables.

Main lessons learned during the execution of this project

The current programing paradigm of the DW2 processor requires either heuristic embeddings or approximate embeddings to implement QUBO problems that do not have connectivity native to the processor. This step leads to a loss of certainty about whether the optimal solution to the original problem is the same as the optimal solution to the embedded problem. This feature potentially results in false negatives when solving decision problems with DW2. For example, if the best answer provided by the processor corresponds to a negative result for the decision problem (e.g., a SAT formula is not satisfiable), yet there exists a better solution to the original decision problem that gives a positive answer (i.e., the SAT formula may be satisfiable). Note that there are no false positives, since a positive answer provides that the corresponding assignment can be checked efficiently.

Optimization problems are better suited for the current programming toolbox. Even though we may not find the optimal solution to a problem, both the heuristic and approximate embedding approaches provide “good solutions”, which can still be very valuable if they can be found faster than with other methods. The heuristic embedding tool developed in this project is designed to generate a sequence of improving solutions, although there is no guarantee that the optimal solution will be found (although additional information about the problem may help identify when optimal solutions are found).

The question of speed up with respect to classical algorithms is very hard to answer. A benchmarking against a particular classical algorithm will not preclude the existence of another, more efficient algorithm. Since we can only estimate the scaling behavior of the runtime performance of the DW2 processor by benchmarking it on a given set of instances, the problem is translated into finding a particular set of instances that show speedup over some set of classical algorithms. Even how to pose the question of speedup has been the subject of intense research. At this point in time, there is no conclusive evidence that the DW2 provides any speedup, but this has only been tested up to 500 variables. New processors with up to 2000 qubits are expected to be available in the next two years.

Although not directly related to this project, very important results have been obtained regarding the quantum nature of the DW2 processor. Even though it is designed to operate in a quantum

mechanical regime, it is not easy to experimentally confirm this feature. During the execution of this project we also performed research aimed at resolving this issue. Two approaches were devised: one provided evidence of a quantum signature by analyzing the statistics of the output of the DW2 processor when solving a carefully designed problem involving 8 qubits. The second approach, implemented in collaboration with the company D-Wave, gave a definitive answer regarding the quantum nature of the device by showing that entanglement is present during the quantum annealing evolution. Whether this entanglement can provide a computational speedup is still an open question.

In terms of the integration of the CEGAR model checking approach and the DW2 processor, the proposed approach was shown to be very straightforward. The obstacles encountered were not related to the fundamental idea of the approach, but rather to the technical limitations of the software tools required (lack of access to the inner workings of the CEGAR implementation available in the different publicly available model checking packages). Any model checking package that provides the required information (i.e., the ILPs to be solved in CEGAR) could be easily integrated to interface with the DW2 processor.

2 INTRODUCTION

The goal of the QCHECK project was to analyze the feasibility of exploiting the computational capabilities of the DW2 adiabatic quantum processor in order to speedup and improve the solution of model checking problems. The DW2 device is designed to solve combinatorial optimization problems by exploiting quantum mechanical effects of an array of Superconducting Quantum Interference Devices (SQUIDs) [1].

One of the main drivers of the computational hardness of model checking problems is the extremely large size of the state space that needs to be considered [2]. The different algorithms and techniques that have been developed to solve model checking problems need to implement in one way or another, a mitigation strategy for this problem. One of the approaches that have been proposed and developed is based on abstractions. The main idea is to replace the system that needs to be checked by an abstraction that has a much smaller state space, with the feature that if a property is found to be true in the abstraction it is automatically true in the original system. Since the size of the abstracted state space is smaller, the algorithms employed to address the abstract problem require much less computational resources.

The abstraction based approach however, comes with a price: a property may be proven wrong in the abstraction when it is actually true in the original system (false negative). To avoid this problem, every counterexample to a property found in the abstraction must be verified as valid, i.e., a corresponding counterexample must exist in the original system. When such a counterexample cannot be produced, we say that the abstraction generated a spurious counterexample, and the truth or falsehood of the property remains unknown.

To solve this issue an approach known as Counterexample Guided Abstraction Refinement has been developed [3]. The basic idea is to use the structure of the spurious counterexample to generate a finer abstraction that would get rid of it. A finer abstraction has a larger state space and so it is important to find a refinement that increases the size of the state space the least. This process continues until the property is proven to hold, or a valid counterexample in the original system is found. We have identified an approach to CEGAR in which combinatorial optimization problems of the form that can be solved by the DW2 processor are a central part of the algorithm: one is to check whether an abstract counterexample corresponds to an actual counterexample in the original system which requires solving an instance of a Boolean Satisfiability problem; the other is at the root of finding the smallest abstraction refinement that can get rid of a spurious counterexample and requires solving and Integer Linear Program.

2.1 D-Wave Two (DW2) adiabatic quantum optimization processor overview

The DW2 adiabatic quantum computer solves a Quadratic Unconstrained Binary Optimization. This optimization consists in finding the vector of binary variables that minimizes the quadratic objective function

$$f(x_1, \ldots, x_n) = \mathrm{Min}_{[\underline{x}]}\{ \sum_{[i<j;\ i=1,\ldots,n;\ j=1,\ldots,i-1]} \boldsymbol{Q}_{ij} x_i x_j \} \qquad (1)$$

where $\underline{\boldsymbol{x}} = (x_1, x_2, \ldots, x_n)$, $x_i \in \{0,1\}$, and $\boldsymbol{Q}_{ij}$ is a matrix of real numbers that determines the objective function. This problem is equivalent (through a simple linear transformation of the variables $x_i \rightarrow s_i = 2x_i - 1$) to the *Ising model*. The *Ising* model represents a set of interacting spin magnets with an energy given by:

$$\boldsymbol{E}\ (s_1, \ldots, s_n) = \boldsymbol{\Sigma}_{[i<j;\ i=1,\ \ldots,\ n]} \boldsymbol{J}_{ij} s_i s_j + \boldsymbol{\Sigma}_{[i=1,\ \ldots,\ n]} \boldsymbol{h}_i s_i \qquad (2)$$

where the spin variables, s_j's, now take the values {+1,-1}, the parameters $\boldsymbol{J}_{ij}$ represent the interactions between two spins, and the parameters $\boldsymbol{h}_i$ correspond to local magnetic fields. Solving the *Ising* model consists in finding the spin configuration that minimizes the energy, $\boldsymbol{E}$. This problem is known to be NP-hard [4], and many important combinatorial problems can be reduced to it [5].

DW2 implements a quantum version of the *Ising* model, where each spin variable is replaced by a Pauli operator σ_z, representing the state of a qubit (quantum bit) that is associated with the magnetic flux of a superconducting quantum interference device (flux-SQUID). The *Ising* Hamiltonian, given by

$$H_{\text{Ising}} = \Sigma_{[i\ =\ 1,..,N]}\ \{\Sigma_{[j=1,\ldots,N]}[\boldsymbol{J}_{ij}\sigma_i^z\sigma_j^z]\} + \Sigma_{[j=1,\ldots,N]} h_i\sigma_i^z \qquad (3)$$

characterizes the quantum mechanical system of spins. The device allows for tunable interactions between the different qubits (i.e., tunable parameters J_{ij}), as well as tunable local biases (parameters h_i).

Quantum annealing in the D-Wave processor proceeds as follows: initially a transverse field is applied such that the lowest energy state has all the spins pointing in the same transverse direction, a quantum superposition of +1 and -1. The parameters are then slowly varied in order to transform the Hamiltonian into H_{Ising}, whose ground state encodes the solution to the optimization problem. The adiabatic theorem of quantum mechanics assures us that, provided this parameter change is slow enough, the final state of the system corresponds to the ground state of the final Hamiltonian [6], i.e., the spin configuration that minimizes the energy function. The values of the spins are obtained by measuring the flux of each qubit at the end of the annealing. In reality, due to the probabilistic nature of this quantum mechanical system, this process must be repeated several times in order to expect to identify the lowest energy configuration.

Our initial D-Wave One (DW1) system had 128 qubits, depicted as green and gray circles in Figure 1. They are arranged in a 4 x 4 array of 8-qubit tiles. In each tile, the qubits are separated in two groups of 4 and connected in a bipartite fashion [7] (each qubit is only connected to all the qubits in the other group). Some qubits in each tile have extra connections to qubits in other tiles, such that the graph is connected (but not fully connected). The connectivity graph is called the Chimera graph [8]. The DW2 processor used in the latter part of the project has 512 qubits. It is composed of a 16 x 16 array of 8-qubit tiles connected in a similar way as in Figure 1.

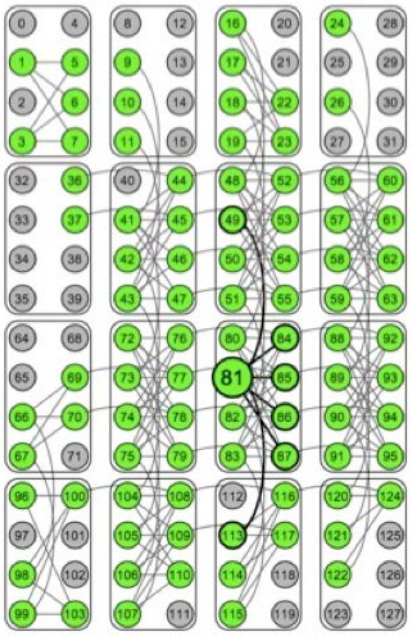

Figure 1. D-Wave One connectivity graph

This connection topology is dictated by constraints imposed by the underlying technology, but the design is scalable up to many thousands of qubits. The lack of full connectivity between all the qubits in the chip prevents a straightforward mapping of an arbitrary *Ising* Hamiltonian (or, equivalently, an arbitrary quadratic function) into the processor. However, although constructing and optimizing this embedding is not a trivial issue, several heuristics have already been developed.

2.1.1 The Physical Principles of the D-Wave Quantum Computer

The basic building block of the DW2 quantum annealing chip is a superconducting flux qubit, (rf-SQUID flux qubit) as depicted in Figure 2. The simplified version consists of two superconducting loops having two Josephson junctions [9]. Each loop is subject to externally-biased magnetic fields (Φ_{1x} and Φ_{2x}) that are used to control the properties of the qubit. The quantum states are associated with the quantized magnetic flux Φ_1.

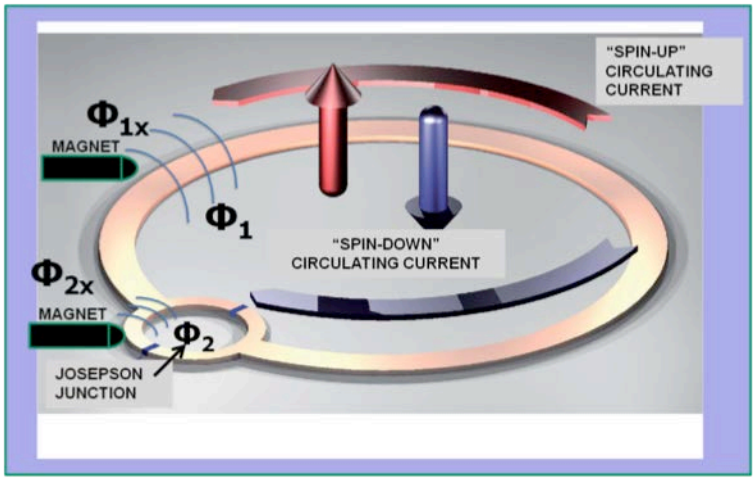

Figure 2. Schematic representation of the compound superconducting loops used to realize the qubits in the D-Wave processor

For low temperatures, it is a good approximation to only consider the two lowest states corresponding to flux pointing up and flux pointing down. The energy profile that describes the system is a double-well potential, represented in Figure 3. The bias fluxes Φ_{1x} and Φ_{2x} are used to adjust the height of the energy barrier δU, and the energy difference between the two states, $2h$. The actual qubits in the quantum computing element chip have extra loops that are used to compensate for undesirable effects due to fabrication manufacturing variations and provide more uniformity in their properties.

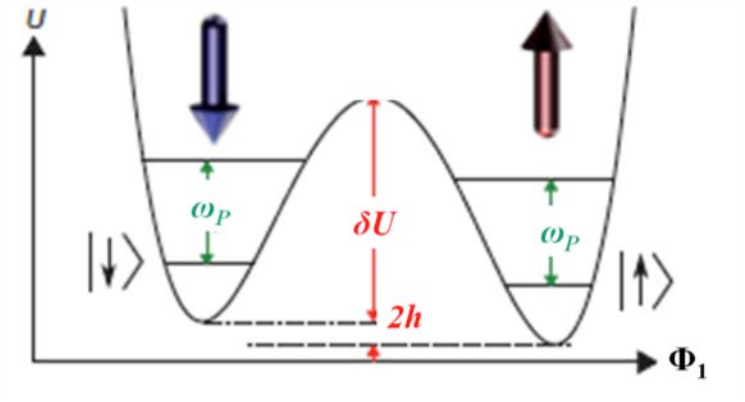

Figure 3. Energy profile of a superconducting flux qubit

2.1.2 Programming and Using the D-Wave Quantum Computer

Programming of the DW2 involves setting the values of the local magnetic fields, and the coupling coefficients for each super-conducting qubit, which determine the desired final (*Ising*)

Hamiltonian. The DW2 is housed in an integrated environment able to be programmed by either desktop computers connected directly to the DW2, or remotely via a local area network, or other remotely-accessed communication networks, to which the desktop computers are connected.

To program the DW2, a user needs to provide the chip with the values of *J* and *h* that satisfy the constraints discussed above. Casting a given problem into the *Ising* form and respecting the constraints on *J* and *h* is part of the "art" of programming DW2. For many discrete optimization problems there are known mappings to the *Ising* model, but those often result in matrices *J* with more connectivity than what is currently available on DW2. One way is to map such *J* into the chip by using certain qubits to simulate more connections, but the price paid is that these qubits are not available to encode the solution. There are also heuristic approaches that aim at approximating a given unconstrained *J* with another matrix *J'* that satisfies the constraints of the chip and has the same minimum of the energy function.

Programming of, and readout from the DW2 occurs through an application-programming interface consisting of function libraries that make calls to the optimization capability of the DW2. These libraries are available in Matlab and Python, and can be used to access the machine's functionalities directly from the programmer's code. These software tools, in conjunction with the machine's circuitry, translate the description of the *Ising* Hamiltonian into the time-dependent classical controlling signals that make the qubits evolve following the required adiabatic evolution. After the qubits are measured, the results are also available through a software interface.

2.1.3 Counterexample guided abstraction refinement

One of the main computational bottlenecks in model checking is related to what is known as the "state space explosion": even for moderately sized systems, the state space needed to describe them is intractably large (a system with 10^{100} states is not uncommon). Developing techniques to deal with this issue is central in model checking. One popular technique is based on the use of Binary Decision Diagrams, a very compact data structure that allows for a succinct description of the state space and the transition system [10]. Another approach is based on abstractions: a smaller system is constructed in such a way that properties proven true in the abstraction are guaranteed to be also true in the original system [11]. The abstraction can then be checked using regular model checking tools (like BDDs for example), which are computationally more efficient since they are applied to a much smaller system.

If an abstraction is not sufficient to prove a given formula, the model checking tool used on the abstraction must provide a counter-example (CE), a path in state-space that violates the formula. This CE can be *real* or *spurious*: a real CE can be mapped to an actual CE in the original (concrete) model, hence disproving the formula; a spurious CE is an artifact of the abstraction and "disappears" when mapped to the original model. To determine which one is the case, we can "simulate" the CE in the original system. This can be posed as finding a satisfying assignment for a Boolean formula. DW2 implements these Boolean *satisfiability* problems by fabricating an energy function that achieves a minimum, when all clauses are satisfied. If the formula is *not satisfiable*, the lowest energy configuration obtained with DW2 will represent an assignment of the Boolean variables that will not satisfy the formula. This can be efficiently checked from DW2's output.

In order to determine if an abstract CE corresponds to an actual CE in the original system (and hence a proof that the property is not satisfied), we need to translate the sequence of transitions that form the abstract CE into a sequence of transitions in the original system. The central question that we need to ask is: *given a transition in the abstract system does a corresponding transition exist in the original system*? It should be remembered that in an existential abstraction, an abstract transition between two abstract states exists, even if only one pair of original states (each mapped to a different abstract state) has a transition (see Figure 4, where the transition between states s_3 and s_6 in the original system induces a transition between the second and third states of the abstraction).

For example, consider an abstract CE, $\hat{T}$, given by a sequence of abstract states $\langle \hat{s_1}, \ldots, \hat{s_n} \rangle$. Given an abstract state $\hat{s}$, the abstraction function h maps states in the original system into the abstract system. The set of states that are mapped into $\hat{s}$ are the ones that satisfy $h(s) = \hat{s}$. If we denote by $R(s_i, s_j)$ the characteristic function of the transitions in the original system (i.e., $R(s_i, s_j) = 1$ if and only if there is a transition between states $s_i$and s_j , and 0 otherwise), then a path $\langle s_1, \ldots, s_n \rangle$ is a concrete representation of the abstract CE $\hat{T}$, if the Boolean formula

$$\Lambda_{i=1}^{n}(h(s_i) = \hat{s_i}) \wedge \Lambda_{i=1}^{n-1} R(s_i, s_{i+1}) \qquad (4)$$

is satisfied, where the first AND operator assures that the states s_i are mapped into the corresponding abstract states $\hat{s_i}$, while the second AND operator assures that there exists a transition in the original system between the states s_i and s_{i+1}. We will show later that finding a satisfying assignment of a Boolean formula can be cast as a 0-1 Integer Linear Program, and this ILP can be mapped into QUBO form implemented by DW2.

2.1.4 Refining the Abstraction

If the CE is spurious, the satisfiability problem presented above identifies an abstract state that causes the violation of the formula being checked. This is due to having clustered together "*dead end*" states (that do not provide a path to an error state) and "*bad*" states (that provide a path to an error state). We can illustrate this behavior with the following diagrams that represent a system and its abstraction (see Figure 4).

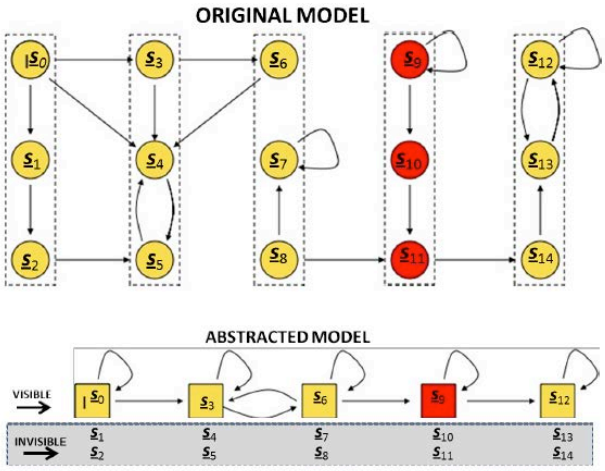

Figure 4. Example of abstracted model

Consider the set of states and their transitions depicted in the original model of Figure 4. Here, $\underline{s}_0$ is the initial state, and the states colored red are error states. Consider a possible abstraction of this system in which all the states inside each dotted-lined rectangle are mapped to the same abstract state. The transition diagram for such an abstraction is given in the lower part of Figure 4. From these diagrams, we can see why a spurious CE may arise. In the original system, it is clear that if we start in the initial state **I**, we will never reach the "error" states (red states in the diagram). However, by looking at the transition diagram for the abstraction, we can see that starting in the abstract initial state we may eventually reach an "error" state. This is easier to see in the diagram of Figure 5.

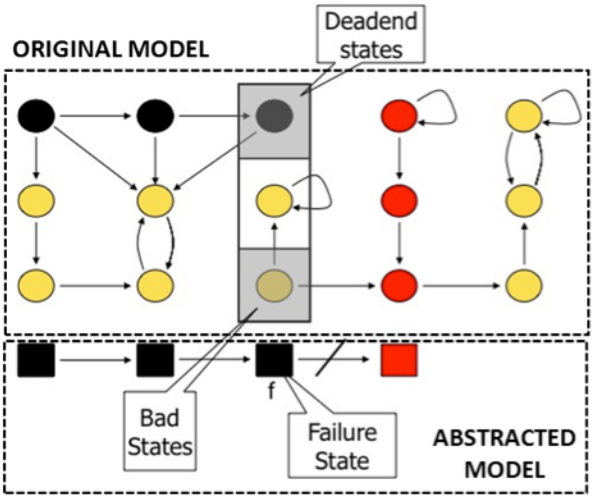

Figure 5. Spurious counter example

The states in black in Figure 5 represent a possible path that the system can take. Note that in the original system that path cannot reach any "error" states. The farthest the system can go is the "dead end" state. However, in the abstraction, the "dead end" state is mapped to the same abstract state as a "bad" state, i.e., a state that can eventually transition into an "error" state, so a spurious transition is created that causes the system to have a valid path from the initial state to an "error" state. In the diagram, we note as a "failure" state the abstract state that is causing a spurious counter-example to appear, because it maps together "dead end" and "bad" states. To refine the abstraction, these states need to be separated and assigned to different abstract states. Therefore, in order to construct viable abstractions, one must follow the steps outlined below:

1. **Minimal Separating Set** – Generate an abstraction where "dead end" and "bad" states are clearly separated. This results in an ILP that the DW2 can solve as an *Ising* problem.
2. **Satisfiability** – Regardless of the approach used to generate abstractions, any CE needs to be verified, if it is real or spurious. This invokes the processing of satisfiability, which is executed in DW2 as an *Ising* model.

The sections below discuss how these steps are formulated and mapped to the DW2 quantum annealer.

2.1.5 Minimal Separating Set

This can be constructed by identifying which of the "invisible" variables should be made "visible" to distinguish "dead end" and "bad" states. The goal is to separate these two sets exactly (i.e., no mistakes allowed) using the smallest number of invisible variables possible, in order to keep the size of the refined abstraction from growing too much. For this, we use the following definitions, assuming that there are 2 sets of states $\boldsymbol{S} = (s_1, s_2, \ldots, s_m)$ and $\boldsymbol{T} = (t_1, t_2, \ldots, t_n)$ that need to be separated ($\boldsymbol{S}$ can represent the "dead-end" states and $\boldsymbol{T}$ can represent the "bad" states). Let $\boldsymbol{W}$ be the set of variables required to specify all states in the original system.

Definition-1: A set of variables $\boldsymbol{U} = (u_1, u_2, \ldots, u_k) \subseteq \boldsymbol{W}$, separates $\boldsymbol{S}$ from $\boldsymbol{T}$ if for each pair of states (s_i, t_j), $s_i \in \boldsymbol{S}$ and $t_j \in \boldsymbol{T}$, there exists a variable u_r such that $s_i(u_r) \neq t_j(u_r)$.

Definition-2: Given 2 sets $\boldsymbol{S}$ and $\boldsymbol{T}$ per Definition-1, find the smallest set of variables $\boldsymbol{U} = (u_1, u_2, \ldots, u_k) \subseteq \boldsymbol{W}$, that separates $\boldsymbol{S}$ from $\boldsymbol{T}$. The set $\boldsymbol{U}$ is called the *minimal separating set*.

We can assign a binary variable $\boldsymbol{v_i}$ to each variable u_i , that will represent whether that variable is included in the separating set or not: if u_i is in the separating set, then $\boldsymbol{v_i} = 1$; otherwise is zero (the corresponding variable is excluded). The conditions are that for each (s_i, t_j) pair at least one of the variables that distinguish between the two states must be selected. Thus, there is a total of *m*x*n* conditions. Under this formulation, the m*inimal separating set* can be solved using integer linear programming in a conventional computer to attain an exact solution.

$$\textbf{Minimize } \boldsymbol{\Sigma}_{[j=1,\ldots,k]}\, \boldsymbol{v_j} \text{ with each } \boldsymbol{v_i} = 0 \text{ or } 1 \qquad (5)$$

$$\textbf{Subject to}: (\forall s \in \boldsymbol{S})\ (\forall t \in \boldsymbol{T})\ \ \boldsymbol{\Sigma}_{[j=1,\ldots,k]}\, \boldsymbol{v_j} \geq 1 \text{ for } s(u_j) \neq t(u_j)$$

Assuming that $\boldsymbol{S}$ represents the "dead end" states and $\boldsymbol{T}$ the "bad" states, the objective function aims to minimize the count on the number of new variables we are including, while the

constraints express the fact that every pair of "dead end"-"bad" states has to be distinguished by *at least one* of the new variables we are including.

We refer now to the case that the sets $\boldsymbol{S}$ and $\boldsymbol{T}$ represent the "bad" and "dead-end" states denoted by $\boldsymbol{S_B}$ and $\boldsymbol{S_D}$. The following lemma from formal methods apply:

Lemma – Let $\boldsymbol{U}$ be the set of variables separating the "dead-end", $\boldsymbol{S_D}$, and "bad", $\boldsymbol{S_B}$, states. Let an abstraction function $\boldsymbol{h'}$ correspond to the visible set $\boldsymbol{V'}$ of variables realizing the abstraction. Also, let $\boldsymbol{V}$ represent the entire set of the original visible variables; then the following applies: $\boldsymbol{V'} = \boldsymbol{V} \cup \boldsymbol{U}$. The abstraction function $\boldsymbol{h'}$ maps $\boldsymbol{S_D}$ and $\boldsymbol{S_B}$ on to different states in the abstract model.

This lemma implies the following: **1)** the number of visible variables have increased from $|\boldsymbol{V}|$ to $|\boldsymbol{V'}|=|\boldsymbol{V}|+|\boldsymbol{U}|$; and **2)** using this augmented set of visible variables the abstraction function will map $\boldsymbol{S}_{\mathrm{B}}$ & $\boldsymbol{S}_{\mathrm{D}}$ to different abstract states, which do need to be considered. As said above, the model checking tool will check if the property of interest is valid in the abstraction. If it is valid, then the procedure is over; if it is not and it generates a spurious CE, the process needs to be repeated until a set of visible variables is identified on which that property holds or a valid CE is found.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Feasibility study of using DW2 for Bounded Model Checking and Binary Decision Diagrams

The analysis of the suitability of DW2 to address BMC problems and to implement BDD-based model checking approaches was a theoretical exercise, and no particular assumptions were required. We will present the details of the problem setup together with the results of this investigation in Section 4.1.

3.2 Benchmarking of DW2 performance on MAX-2-SAT against classical solver MaxWalkSAT

The main goal of this task was to benchmark the performance of the DW2 processor on a native problem, against a classical heuristic solver. The rational for this task was that benchmarking up to that point was performed against classical exact solvers, in particular AK-MAXSAT [12]. Since this solver provides a guarantee of optimality, it requires more resources (i.e., run time). The DW2 processor is a probabilistic solver (solution is provided with a finite probability and no optimality guarantee is given), and so it is not fair to compare it with exact solvers.

3.2.1 MAX-2-SAT

The choice of MAX-2-SAT is based on the fact that this problem can be trivially written as an Ising problem that is native to the DW2 processor.

Definition of MAX-2-SAT: given a Boolean formula in conjunctive normal form with 2 literals per clause, find the maximum number of clauses that can be simultaneously satisfied.

The key point in implementing this problem using the DW2 processor is to notice that for each 2-literal clause, we can construct a Hamiltonian of Ising form whose ground state is composed by the satisfying assignments for the clause. For example, consider the 2-literal clause ($x_i \vee x_j$). This clause is satisfied if any of the two variables is TRUE. To map this problem into an Ising form we will associate to each variable the state of a qubit, with x = TRUE $\rightarrow |+1\rangle$ and x = FALSE $\rightarrow |-1\rangle$. Consider then the 2-qubit Hamiltonian

$$\boldsymbol{H} = \frac{\mathbf{1}}{\mathbf{4}}(\mathbf{1} - \boldsymbol{\sigma_i} - \boldsymbol{\sigma_j} + \boldsymbol{\sigma_i} \otimes \boldsymbol{\sigma_j}) \qquad (6)$$

where the $\boldsymbol{\sigma_i}$ is the Pauli operator associated with qubit i, and $\{|+1\rangle , |-1\rangle\}$ are its corresponding eigenvectors. Table 1 shows the truth table of the 2-literal clause, and the energies of the associated states.

Table 1. Truth table for Boolean OR and energies of associated 2-qubit Hamiltonian

x_i	x_j	$x_i \vee x_j$	$\boldsymbol{\sigma_i}$	$\boldsymbol{\sigma_j}$	$\boldsymbol{H}$
F	F	F	-1	-1	1
F	T	T	-1	+1	0
T	F	T	+1	-1	0

T	T	T	+1	+1	0

We can see that all satisfying assignments are associated with states of energy 0, while the only non-satisfying assignment corresponds to a state with energy 1. Hence, the ground state of the Hamiltonian H is composed exactly by all the satisfying assignments of the 2-literal clause. If any of the variables appear negated in the formula, we just need to flip the sign of the corresponding Pauli operator on the Hamiltonian.

If we have a conjunction of many clauses $C_1 \wedge \ldots \wedge C_M$, we just need to add the corresponding Hamiltonians $H = H_1 + \ldots + H_M$. For any possible truth assignment to the Boolean variables, the energy of the state will be increased by 1 for every unsatisfied clause. Then the energy of the ground state of H will be the *minimal* number of unsatisfied clauses, from which we can trivially infer the *maximum* number of satisfied clauses, i.e., the objective of the MAX-2-SAT problem. It is then clear that we can look at MAX-2-SAT as a native problem to the DW2 processor. Even though the decision problem 2-SAT is known to have a polynomial-time solution, the optimization problem MAX-2-SAT is NP-hard (i.e., a polynomial-time algorithm for it would imply the existence of a polynomial-time algorithm for all problems in NP).

3.2.2 MaxWalkSAT

We chose the MaxWalkSAT [13] solver as the classical algorithm for the benchmarking. This solver applies heuristic methods to provide an approximate solution to a MAX-2-SAT instance. Since it is not required to provide any guarantees of optimality (as exact solvers do) it can run much faster on many instances. We considered that this provided a better comparison between classical solvers and the DW2 processor.

MaxWalkSAT is a variant of WalkSAT, a heuristic SAT solver. In its more general form, MaxWalkSAT solves the weighted SAT problem, in which each clause is given a weight and the goal is to maximize the total weight of all simultaneously satisfied clauses. In our case, we set the weights to 1. The algorithm for WalkSAT starts with a random truth assignment for all the variables, then randomly selects an unsatisfied clause and a variable within that clause is flipped. This variable can be chosen either at random, or as the variable whose flipping minimizes the number of already satisfied clauses becoming unsatisfied. So in a sense, it is a mixture of deterministic local search and random jumps.

3.2.3 Instance ensemble

In order to implement this comparison we generated MAX-2-SAT instances that were native to the architecture of the DW2 processor. We generated problems with N variables, for N = 20,40,…,500. The number of clauses was chosen to be 2N, since it is known that this ratio of clauses to variables generates instances that are typically hard to solve.

The ensemble was composed of 1000 instances for each value of N. The instances were constructed in the following way. For each value of N, we chose N qubits that formed a connected subset of the processor (to avoid assigning variables to qubits that were not connected to other qubits of the set). Then we randomly picked M=2N of the available couplers associated with the set of qubits to represent the 2-literal clauses. Finally, for each clause we randomly (probability ½) negated the literals. This construction assured us that all clauses were distinct, and hence the total number of clauses was indeed 2N.

3.2.4 Benchmarking strategy

To compare the performance of the classical and quantum solvers we implemented the following strategy. First, by using the exact solver AK-MAXSAT, we found the optimal value of the objective function for every instance in the ensemble. This value was later used to estimate the probability of success of each solver on each given instance.

Quantum solver (DW2): we ran each instance a thousand times, using an annealing cycle of 20 microseconds. We compared the value of the objective obtained in each run with the known optimal value, and used this information to compute the probability of success for each given instance. Then we used this information to compute the expected number of repetitions (or runs) needed to obtain the optimal value at least once with at least 99% probability. This number of repetitions times the annealing time used (20 microseconds) was the performance figure we used for each instance. We then averaged this value over all instances with the same number of variables N, and used it to compare with the classical solver.

Classical solver (MaxWalkSAT): the classical algorithm MaxWalkSAT requires another input parameter called the "cutoff", that gives an upper limit on the number of iterations performed before stopping. Clearly, if the cutoff is small, the algorithm will be faster but we may not find the optimal solution. On the other hand if the cutoff is very large, the algorithm will take more time but will have a better chance of finding the optimal solution. There is then a tradeoff between the value of the cutoff and the time it would take the algorithm to find the optimal solution with probability at least 99%. We ran every instance with different values of the cutoff in order to find a value that will reach the optimal solution with 99% probability in the fastest run time. We then averaged these values over all of the instances with the same number of variables N. All the instances were run on a Mac Pro with a 2.6 GHZ processor and 48Gb of RAM.

3.3 Development of heuristic embedding algorithm

The main goal of this task was to develop a tool that would allow us to embed problems that do not match the processor's connectivity. As discussed in Section 2, the DW2 processor has a very particular connectivity graph called the Chimera graph that is the result of design compromises between scalability and algorithmic power. The connectivity graph is sparse, and each is qubit connected to at most 6 other qubits.

This design feature has an impact on the type of problems that can be embedded in the processor. A general Ising model will have an underlying graph of couplings, and if this graph is not a subset of the Chimera graph we need to implement alternative ways of embedding the problem. Even if a given instance was a subgraph of the Chimera graph, finding the appropriate mapping is an instance of Subgraph Isomorphism, another combinatorial optimization problem that may be as hard as the original Ising instance. Hence, with the current design of the processor we have no choice but to develop alternative methods to embed problems. It is important to point out that this issue is not particular to the D-Wave processors. For any implementation of adiabatic quantum optimization, the connectivity of the processor will be associated with some physical interaction between qubits. These interactions tend to be local and thus require that the qubits are close to each other. This will put a limit to the number of interactions a given qubit can represent, since the number of local neighbors in any reasonable architecture will be limited and much smaller than the total number of qubits. Hence the problem of embedding is central to the adiabatic quantum optimization approach and not just a D-Wave issue.

Since most Ising problems of interest will not fit directly into the Chimera graph, it is necessary to develop techniques to go around this issue. We have already mentioned that an exact embedding requires solving a hard problem and does not seem to be a scalable solution. Hence, we need to apply some kind of *approximate embedding*. This requires discarding some of the information (i.e., the couplings) that defines the problem in order to generate a related Ising problem that can be fit into the Chimera graph. This approach has been used by Google and D-Wave in an image recognition application [14], where they reduced the training of a strong classifier to a QUBO problem. Their approximate embedding scheme followed a greedy algorithm, that aimed at keeping the largest couplings (in absolute value) with the rational that these couplers will be more important in determining the structure of the best solutions. It is important to note that this step requires a certain preprocessing of the input instance that increases the computational resources required. Also, there are no theoretical results that would guide this process or give any guarantees on the quality of the solutions obtained.

3.3.1 Iterative heuristic embedding

In order to address the drawbacks of the approximate embedding method, we considered a different approach that aims at taking advantage of the sampling capabilities of the DW2 processor. The idea comes from an approach to optimization problems known as "Probability collectives" [15]. The main idea is to replace an optimization problem with a sampling problem. Given an objective function over binary strings G(x), one approach to find its minimum will be to sample from its Gibbs distribution, which is given by

$$P(\mathbf{x}) = \exp(-\beta * G(\mathbf{x})) / Z \qquad (7)$$

where beta is the inverse temperature, and Z is the partition function, which is defined as $Z = \Sigma \exp(-\beta * G(\mathbf{x}))$, where the sum runs over all binary strings. It is clear that this distribution is biased towards configurations that have a small value of $G(\mathbf{x})$ due to the exponential factor. So if we had access to a machine or algorithm that generated samples following this distribution, with high probability we would obtain the minimum configuration.

The key point is to consider the DW2 processor as a *parameterized sampler*, where the parameters are the local fields and the couplings (h_i,J_{ij}), and the output is a distribution over the set of binary strings. The goal is then to find a set of parameters that produce an output distribution that is "close" to the Gibbs distribution associated with the objective function $G(\mathbf{x})$. This sets up an iterative procedure:

1. Initialize the set of parameters (h_i,J_{ij}).
2. Sample the output of the DW2 processor using these parameters.
3. Compute measure of "closeness" between this output distribution and $P(\mathbf{x})$.
4. Update the parameters (h_i,J_{ij}) in order to decrease the measure of "closeness".
5. Go back to Step 2.

This process continues until a termination criterion is reached. Every time we sample the processor, we can compute the value of the objective function G on all samples and keep track of the one that gives us the minimum.

As a measure of "closeness" we chose the relative entropy (or Kullback-Leibler divergence) between the two distributions [16]. This measure has the property of being non-negative, and vanishing if and only if the two distributions are identical. If we call $Q(\mathbf{x}; h_i,J_{ij})$ the output distribution of the quantum processor (that depends on the parameters (h_i,J_{ij})), the relative entropy between Q and P is defined by

$$D(Q||P) = \Sigma\, Q(\mathbf{x}; h_i,J_{ij}) \log (Q(\mathbf{x}; h_i,J_{ij}) / P(\mathbf{x})) \qquad (8)$$

where the sum is taken over all binary strings. Our goal then is to find the values of the parameters (h_i,J_{ij}) that minimize the relative entropy. This is an optimization problem of a continuous function over a set of continuous variables, so we chose a gradient descent method.

In order to compute the components of the gradient of the relative entropy, we would need to know the functional form of $Q(\mathbf{x}; h_i,J_{ij})$. However, this functional form is not available to us, and we can only sample from the distribution $Q(\mathbf{x}; h_i,J_{ij})$. To move forward, we made the assumption that $Q(\mathbf{x}; h_i,J_{ij})$ was the Gibbs distribution associated with the Ising energy, that is

$$Q(\mathbf{x}; h_i,J_{ij}) = \exp(-\beta\, E_{Ising} (\mathbf{x}; h_i,J_{ij})) / Z_Q \qquad (9)$$

where $Z_Q(h_i,J_{ij}) = \Sigma\ \exp(-\beta * E_{Ising} (\mathbf{x}; h_i,J_{ij}))$ is a normalization constant that depends on the parameters (h_i,J_{ij}), and the sum is over all binary strings. By making this assumption we can explicitly compute the components of the gradient and obtain

$$\nabla_J = -\beta\, \{\langle (2x_i - 1)(2x_j - 1) \log(Q(\mathbf{x}; h_i,J_{ij}) / \exp(-\beta * G(\mathbf{x})))\rangle - \langle (2x_i - 1)(2x_j - 1)\rangle \langle \log(Q(\mathbf{x}; h_i,J_{ij}) / \exp(-\beta * G(\mathbf{x})))\rangle \}$$

$$\nabla_h = -\beta\, \{\langle (2x_i - 1) \log(Q(\mathbf{x}; h_i,J_{ij}) / \exp(-\beta * G(\mathbf{x})))\rangle - \langle (2x_i - 1)\rangle \langle \log(Q(\mathbf{x}; h_i,J_{ij}) / \exp(-\beta * G(\mathbf{x})))\rangle \} \qquad (10)$$

The expectation values that appear in the gradient are taken with respect to the distribution $\exp(-\beta\, E_{Ising} (\mathbf{x}; h_i,J_{ij})) / Z_Q$, i.e., the Gibbs distribution associated with the Ising model implemented on the processor. Even though we know it's functional form, this expression is hard to compute because it requires summing over all binary strings to obtain the normalization constant Z_Q, and this sum has exponentially many terms, making it impractical for large problems. In order to get around this obstacle, we will make another approximation and use the sample averages to compute the expected values. The sample averages can be obtained by evaluating the expressions on the samples produced by the processor. Since we will only generate a fixed number of samples, this computation can be done efficiently.

Note that the algorithm makes two approximations: first, it assumes that the output distribution from the DW2 processor is a Gibbs distribution in order to compute the gradient of the relative entropy; and second, it replaces the expected values over this Gibbs distribution by the sample averages.

3.4 Integration of CEGAR approach with DW2

The CEGAR algorithm is a means of tackling the state space explosions that often arise in model-checking. In CEGAR, one initially computes an *abstraction* of the original model that can be model-checked more easily than the full model. This must be an abstraction that is *conservative*, in a sense we describe below. One then checks the abstracted model to see if the property holds in the abstracted model. If it holds, we are done; the system passes the test. Here is where the conservative nature of the abstraction is critical: it must be the case that if the system passes the check the property is, in fact, safe (the check must be *sound*); however, CEGAR admits false positives (where the check fails, although the system is safe – the check is not *complete*). Typically the CEGAR algorithm is applied to reachability problems, where the safety property states that the system must not reach some undesirable state. A conservative abstraction is used which increases the set of reachable states, so that the check will be sound.

If we find an abstract counterexample, we commence the part of the process that gives the algorithm its name. First we must *check* to see if the counterexample is sound. We do this by "replaying" the counterexample in the full model, instead of the abstraction. If the counterexample is found to be sound, we are done: the system is unsafe, and must be corrected. On the other hand, if the counterexample is unsound, we must *refine* the abstraction and repeat the process. A simple diagram of the CEGAR procedure is presented in Figure 6. The abstraction refinement is counter-example guided in the sense that we find a place in the counter-example trace where the abstract counterexample cannot be followed. In this case, what must have happened is that the abstract counter-example progresses from abstract state as_i to as_{i+1} but there is no way to progress from a corresponding concrete state $h^{-1}(as_i)$ to $h^{-1}(as_{i+1})$ ($h^{-1}$is the inverse of the abstraction, so $h^{-1}(as_i)$ is the set of concrete states that correspond to the abstract state as_i.) To refine the abstract state, we find the set of concrete states that satisfy the description of abstract state i, $h^{-1}(as_i)$, and the set of concrete states from which $h^{-1}(as_{i+1})$ is reachable, and refine by adding state features that separate these two sets. The first set of states is called the "dead end states," and the second set is called the "bad states."

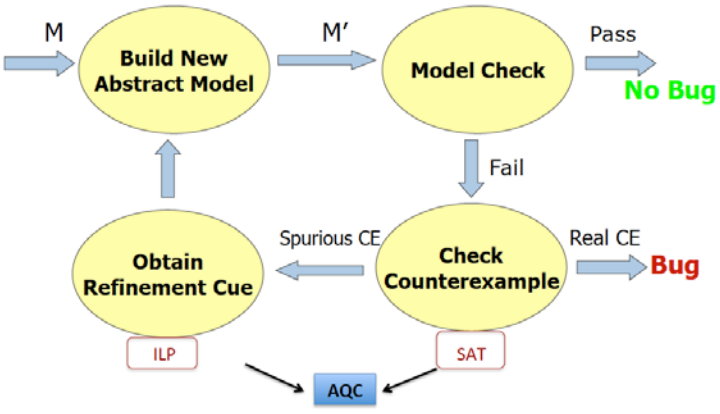

Figure 6. CEGAR loop

In the QCHECK project, we experimented with applying the D-wave quantum computer to two steps of the CEGAR process:

1. Checking the soundness of the abstract counter-example and
2. Finding an (approximately) optimal refinement to separate the dead-end from the bad states.

The first step involved solving a SAT problem that encoded the existence of an actual counter example associated with the abstract counterexample. The second involved solving an ILP in order to find the smallest set of hidden variables that needed to be made visible in order to get rid of the spurious counterexample. These two problems (SAT and ILP) are special cases of combinatorial optimization problems, and can both be cast as QUBO problems that can be solved with DW2. Actually, SAT can be cast as a particular instance of ILP as we show below.

3.4.1 Converting SAT to ILP

In a Boolean satisfiability problem, we have a set of Boolean variables $\{\boldsymbol{x_i}\}$, and a set of clauses formed by combining a number of those variables and their negations with the logic operator OR. For example, a clause can take the form

$$C = \boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \overline{\boldsymbol{x_3}} \qquad (11)$$

where $\vee$ stands for the OR operator and $\overline{\boldsymbol{x_3}}$ stands for the negation of $\boldsymbol{x_3}$. A clause is satisfied if at least one of its Boolean operands is TRUE. The Boolean satisfiability problem consists in finding an assignment of truth-values for all the variables such that all clauses are satisfied. As we have shown above, a step in the abstraction refinement process requires us to solve a Boolean satisfiability problem in order to determine if an abstract CE corresponds to a CE in the concrete system. We will now show how to map this problem into an ILP. For each clause, we will replace the OR operator by the "+" (sum) operator, every Boolean variable $\boldsymbol{x_i}$ by a binary variable x_i if it is not negated, and by (1- x_i) if it is negated, and we will impose the condition that this sum must be greater or equal to 1. We will use the numerical value 1 to represent the Boolean value TRUE, and 0 to represent FALSE. Hence, the clause C defined above will be transformed into

$$\boldsymbol{x_1} + \boldsymbol{x_2} + (\mathbf{1} - \boldsymbol{x_3}) \geq \mathbf{1} \qquad (12)$$

It is not difficult to see that the logical clause C is TRUE, if and only if the above linear inequality is satisfied. Then, satisfying a set of clauses is equivalent to checking the feasibility of satisfying a set of linear inequalities like the one above (one for each clause). Since the variables are binary, this is nothing but a particular case of an Integer Linear Program, one in which there is no linear function to optimize and the variables are restricted to be 0 or 1. This type of problem can be implemented on DW2 as explained in the next section.

3.4.2 Converting ILP to QUBO

ILPs are known to be computationally expensive for conventional computers (since they are NP-hard). Nevertheless, we can map this problem to an *Ising* model that can be implemented natively on the DW2 quantum processor. Let us show in general how this mapping can be done. A typical binary integer linear programming problem can be written as:

$$\textbf{minimize}_{[\underline{x}]}\ \Sigma_{[j=1,\ldots,n]} c_j x_j$$
$$\textbf{subject to}:\ \Sigma_{[j=1,\ldots,n]} B_{kj} x_j \geq b_k\ ;\ b_k \geq 0\ ;\ 1 \leq k \leq q \qquad (13)$$

where each x_j is a Boolean number taking the value of 0 or 1, $x_j \in \{0,1\}$, all the coefficients, c_j, B_{kj}, are integer numbers (positive or negative), and b_k is non-negative integer number. The way to deal with these constraints is to transform them into a quadratic penalty term that will increase the energy of the *Ising* model when they are not satisfied. This is easily done with equality constraints. For the inequality constraints, we first transform them into equality constraints by adding extra "*slack*" variables (this is a canonical way of dealing with inequalities in optimization problems). The detail we have to take into account is that the variables that we add must be binary too (as the x_j's). For the k^{th} inequality constraint, we define $m_k = \Sigma_{[i=1, \ldots, n]} B_{ki}$ including only the B_{ki}'s that are non-negative ($B_{ki} \geq 0$). In essence, m_k represents the maximum numerical value of the sum, since each x_j is a Boolean number. The role of the slack variables is to add whatever value is needed in order to transform the inequality constraint into an equality constraint. The slack variables must construct a non-negative integer number p_k such that

$$\Sigma_{[j=1,\ldots, n]} B_{kj} x_j - p_k = b_k \qquad (14)$$

In order to satisfy this equality, the integer number p_k should be allowed to be as large as the maximum difference between $\Sigma_{[j=1,\ldots, n]} B_{kj} x_j$ and b_k; this constitutes its numerical range. This maximum difference is just $|m_k - b_k|$. The most efficient way (in terms of using the smallest number of extra slack variables) is to express the non-negative integer p_k by an expansion in powers of 2 (binary expansion). Therefore,

$$p_k = \Sigma_{[j= 1, \ldots, Dk]} 2^{j-1} \beta_j \qquad (15)$$

where each β_j is a binary variable, $\beta_j \in \{0.1\}$, and $D_k = [|\log_2(m_k - b_k)|]+1$, since $p_k \leq |(m_k - b_k)|$. This is a consequence of the binary arithmetic stating that the number of bits required to express a positive number of magnitude $\leq N$ is equal to $\log_2 N + 1$. Here, the number of bits required (and hence, the number of binary slack variables) will be D_k, and the square brackets, […], represent the largest integer smaller than the argument. So we define new slack binary variables β_{ki} that transform the inequality constraints in Equations (13) to equality, namely,

$$\Sigma_{[j=1,\ldots, n]} B_{kj} x_j - \Sigma_{[j=1, \ldots, Dk]} 2^{j-1} \beta_j = b_k \; ; \; 1 \leq k \leq q \qquad (16)$$

It should be clear that if this equality is satisfied, Equation (16), then, the inequality in Equation (13) is also satisfied. To generate an *Ising* problem we add, to the linear objective function of the binary ILP, Equation (13), the square of the difference of the left-hand-side and the right-hand-side of each equality, Equation (16), times a penalty constant $K>0$, and we now take the minimum over the original variables x_i and the new slack variables β_{kj}. We finally get:

$$\mathbf{min}_{[\underline{x},\underline{\beta}]} \{ \Sigma_{[j=1,\ldots, n]} c_j x_j + K \Sigma_{[k=1,\ldots, q]} ((\Sigma_{[j=1,\ldots, n]} B_{kj} x_j - \Sigma_{[j=1, \ldots, Dk]} 2^{j-1} \beta_{kj}) - b_k)^2 \} \qquad (17)$$

where $\underline{x} = \{x_i; i=,..,n\}$, $\underline{\beta} = \{\beta_{kj}; j=1,.., D_k; k=1,.., q\}$, and $D_k = [|\log_2(m_k - b_k)|]+1$ for k=1,..,q.

Equation (17) has the form of an *Ising* model (a linear term and a quadratic term). We can make this more explicit by expanding this expression and grouping the corresponding terms. In order to make things more compact we define the following new integer matrices:

$$\Phi_{ik} \begin{cases} = B_{ik} & \text{if } 1 \leq i \leq n \text{ and } 1 \leq k \leq q. \\ = 2^{i - (n + \Sigma[l= 1, \ldots, k-1] D l)} & \text{if } (n + 1 \leq i \leq (n + \Sigma_{[l=1, \ldots, k-1]} D_l) \text{ and } 1 \leq k \leq q. \\ = 0 & \text{Otherwise} \end{cases} \qquad (18)$$

and

$$g_i \begin{cases} = c_i & \text{if } 1 \le i \le n \\ = 0 & \text{if } (n+1 \le i \le (n + \Sigma_{[k=1,\ldots,q]} D_k) \end{cases} \quad (19)$$

where $\Phi = \{\Phi_{ik};\ k=1, \ldots, q;\ i=1, \ldots, n + \Sigma_{[k=1,\ldots,q]} D_k\}$ is a qx $(n + \Sigma_{[k=1,\ldots,q]} D_k)$ matrix, and $\underline{\boldsymbol{g}}$ is a $1x(n + \Sigma_{[k=1,\ldots,q]} D_k)$ row vector. Also, we augment the original vector $\underline{\boldsymbol{x}}$ to a new vector $\underline{\boldsymbol{z}}$, by appending the slack variables $\boldsymbol{\beta}$, namely creating a new $(n + \Sigma_{[k=1,\ldots,q]} D_k)x1$ column vector , $\underline{\boldsymbol{z}}$, $\underline{\boldsymbol{z}} = \{x_1,\ldots,x_n, \beta_{kj}\ ;\ k = 1,\ldots,q;\ j = 1, \ldots, D_k\}$.

Equation (17) can be transformed into an *Ising* model solvable by the DW2 quantum computer. The minimization is over the augmented vector, $\underline{\boldsymbol{z}}$:

$$\mathbf{min}_{[\underline{z}]} \{ \underline{\boldsymbol{z}}^T(\Phi^T\Phi)\underline{\boldsymbol{z}} + (\underline{\boldsymbol{g}} - 2\underline{\boldsymbol{b}}^T\Phi)\underline{\boldsymbol{z}} \} \quad (20)$$

where $\underline{\boldsymbol{b}}$ is the qx1 column vector $\{b_1, \ldots, b_q\}$.

In summary, this ILP-based approach to abstraction refinement can be cast as an *Ising* problem that can be natively implemented on the D-Wave quantum computing processor.

3.5 Implementation of Model Checking example

The final task of this project consisted of applying the tools and techniques developed in the previous tasks to a simple model checking example. We considered different systems and finally converged on a model that was inspired by an avionics example problem, but did not correspond to any real hardware or software. It was a toy model designed to show a proof of concept for the integration of DW2 and the CEGAR approach to model checking, and was tweaked to have certain features that would result in non trivial problems for the DW2 processor to solve. A more detailed description of the example will be provided when discussing the result of the actual implementation in the following section.

4 RESULTS AND DISCUSSION

4.1 Feasibility study of using DW2 for Bounded Model Checking and Binary Decision Diagrams

Bounded Model Checking is an alternative approach to model checking. Its main feature is that properties are checked to hold for a finite number of time steps. In this way, the properties can be expressed as Boolean formulas and the algorithm consists in applying a satisfiability solver to determine if the formula has a satisfying assignment: if it does, the property is proven true, and if it does not, it is proven false.

This approach is different from the more general model checking approach, whose algorithms are based on constructing a sequence of sets that tend to the set of all states that satisfy a particular temporal logic formula. The advantage of the general approach is that properties can be proven to hold for all possible executions of the system, while BMC are only restricted to executions within a finite temporal horizon. The price paid in BMC is a lack of completeness, i.e., it is not possible to prove or disprove every formula in a given temporal logic. However, there are certain properties that can be proved and others that can be disproved. In particular, BMC is well suited for finding short counterexamples, so its goal leans more towards finding bugs than proving correctness. In this area, BMC can be more efficient than general model checking techniques based on BDDs.

BMC proceeds in two steps: first, a finite length execution path satisfying a certain property on the space state is encoded as a propositional formula; then, a satisfiability solver is applied to find a satisfying assignment or prove none exists. If a satisfying assignment is found, it can be decoded to represent a particular path on the state space that satisfies the property. Depending on the property being considered, this could be a proof of a liveness property (i.e., a state with a certain property can actually be reached), or a counterexample that disproves a property (by showing a specific path that violates it). It is in this second step of BMC that we believe DW2 can provide an advantage, since Boolean satisfiability is a decision problem that can be cast as the type of combinatorial optimization problem that DW2 is designed to solve.

4.1.1 Creating propositional formulas in BMC

In BMC we consider three elements:

1. A transition system *M*.
2. A temporal logic formula Φ.
3. A time bound *k*.

From these three elements we construct a propositional formula that checks the satisfiability of the property represented by Φ in the transition system *M* for paths of length at most *k*. We can construct the *unrolled transition relation* defined by

$$[M]_k := I(s_0) \wedge \bigwedge_{i=0}^{k} T(s_i, s_{i+1}) \qquad (21)$$

where $I(s_0)$ is the characteristic function of the set of initial states, and $T(s_i, s_{i+1})$ is the characteristic function of the transition relation. Basically, $[M]_k$ is the set of all allowed paths of

length *k* that start in the set of initial states. We can also construct $[\Phi]_k$, which is a formula that will be true if and only if Φ is valid along a path of length *k*. In BMC, we want to find whether the conjunction formula $([M]_k \wedge [\Phi]_k)$ has a satisfying assignment.

As an example, consider the Computation Tree Logic (CTL) formula **EF** *p*, that means that a state satisfying the propositional formula *p* is reachable from the initial state. Applying the BMC approach to this formula for *k=2*, results in the following formula:

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge (p(s_0) \vee p(s_1) \vee p(s_2)) \qquad (22)$$

where $[\mathbf{EF}\ p]_2 = (p(s_0) \vee p(s_1) \vee p(s_2))$. Solving this formula consists in finding a sequence of three states (s_0, s_1, s_2) that satisfies it, or proving that no such satisfying assignment exists.

4.1.2 Mapping of propositional formulas for BMC into DW2

Our approach to implementing a satisfiability solver using DW2 is based on a transformation of SAT to a 0-1 ILP, which in turn can be transformed into a QUBO problem. The transformation from SAT to 0-1 ILP assumes that the SAT formula is given in conjunctive normal form (CNF), that is

$$F = C_1 \wedge \cdots \wedge C_n \qquad (23)$$

where each clause C_i is a disjunction of literals. The formulas generated in the BMC approach need not be of this particular form. Even though it is a conjunction of three elements (the characteristic functions of the set of initial states and the unrolled transition, together with the formula specifying the particular property being considered) each of these terms will not necessarily be just a disjunction of literals.

It will then be necessary to transform them into CNF before we can map the problem into the chip. This type of preprocessing is also present in classical implementations of BMC, and researchers have developed subroutines that perform this translation. The drawback is that they usually require the addition of extra variables, which will reduce the number of qubits available to represent state variables, resulting in a reduction of the size of the systems we will be able to analyze. Given that the number of qubits available is fixed, and even though it is expected that this technology will scale with time it will only do it moderately, it is clear that preprocessing techniques and hybrid approaches that allows us to break the problem into smaller ones will be an extremely important component of any application of quantum computing in the adiabatic model implemented by D-Wave devices.

4.1.3 Results of the suitability study for Bounded Model Checking

The analysis presented above shows that the BMC approach is indeed suitable for implementation using the capabilities of DW2. Since the problem reduces to SAT and SAT can be cast as a combinatorial optimization problem of the form solved by DW2, the main issues to address are how to efficiently use the resources of DW2, namely how to encode the SAT formula in a way that it makes the most efficient use of the qubits available. These are not different from the ones that we had already anticipated when we proposed to use DW2 to check if the abstract counter examples obtained within the CEGAR framework are associated with real counter examples in the original system (this is again an instance of SAT).

4.1.4 Implementation of Binary Decision Diagrams using DW2

The purpose of this analysis was to study whether techniques based on BDDs used in model checking of finite discrete systems can be cast as an optimization problem that would be suited for implementation with DW2. We will first discuss the basic features of BDDs, why they are useful in MC, and the issues that arise when we try to implement BDD based algorithms as combinatorial optimization problems.

4.1.5 Binary Decision Diagrams

BDDs are a type of data structure used to represent Boolean functions. A Boolean function is just a function from $\{0,1\}^N$ to $\{0,1\}$, that is, a function from the set of all N-bit strings to $\{0,1\}$. We can interpret these functions as the characteristic function of a given subset to N-bit strings (the subset that evaluates to 1). Since the model checking problem is essentially a set problem (i.e., whether the set of initial states **I** is included in the set of states that satisfies a formula **Φ**), BDDs are a natural tool for model checking algorithms.

Why are BDDs so useful in MC? The main problem of MC is the state space explosion. BDDs provide a structure that in some cases can represent both the subsets of the state space and the state transition relation in a compact way. Furthermore, we can define operations on BDDs that implement the set operations required by the model checking algorithms (unions, intersections, etc.) This allows the algorithms to operate not with the subsets, but with their characteristic functions. The point is that a subset can have an exponential size (in the number of variables used to describe it) while its characteristic function can be described much more compactly. In this way, the algorithms proceed in a way that it does not require an explicit description of an exponentially large set at any point.

Formally speaking, a BDD is an acyclical directed graph, with one root node, two terminal nodes, and a set of internal nodes that have one predecessor and two successors. Each node represents a variable, and the value of the Boolean function the BDD represents is obtained by traversing the graph starting from the root node, and moving to the successor node associated with the value of the variable the node represents; the terminal node that is reached is the value of the function for that particular bit string.

One of the useful features of a BDD is that, once a variable ordering is fixed, the BDD can be written in a unique canonical way, which makes comparing two BDDs (i.e., two subsets of N-bit strings) computationally simple for polynomially sized BDDs. The size of this canonical BDD depends crucially on the variable ordering, and two different variable orderings can result in two BDDs of exponentially different size. Even though finding the best variable ordering for a BDD is itself a hard computational problem, heuristics have been developed that result in reasonable sized BDDs for many problems studied in practice.

Another important reason for working with canonical BDDs is that the set operations required by model checking algorithms can be implemented directly on the BDD by applying a small set of graph operations: given two BDDs in canonical form, we can compute the canonical BDD corresponding to logical operations applied to them (like AND and OR) in a very compact way. To understand the real practical impact of this, we need to look at how model checking algorithms work.

4.1.6 Model checking algorithms

A typical model checking problem consists in proving that a given property is satisfied for a certain set of initial states. The property will depend on the type of logic that we are considering, but typical examples are whether a state with a certain characteristic is eventually (or always) reached, or that a certain set of states is never reached. The essence of model checking algorithms is to compute the set of states that satisfy the property and then check if the set of initial states is included in this set. The advantage of using BDDs for these algorithms is that these sets are never defined explicitly (they could be exponentially large).

As an example of the basic structure of these algorithms, let us consider a path formula in CTL of the form $(\Phi \text{ U } \Psi)$, which means that for all paths property Φ holds until property Ψ becomes true. The goal is to find the set of all states that satisfy this formula. The algorithm will proceed iteratively and generate a nested sequence of sets that satisfy the formula, until the set obtained does not change: this will be the set of all states **S** that satisfy the formula, and then we can easily check if the set of initial states is included in it.

The steps in such an iterative algorithm will be something like this:

1. Let $\mathbf{S_1}$ be the set of states that satisfy Ψ (sets like this are assumed to be provided). Then $\mathbf{S_1} \subseteq \mathbf{S}$, since all states that satisfy Ψ trivially satisfy $(\Phi \text{ U } \Psi)$.
2. Next, find the set of states that both satisfy Φ, and such that there is a transition from it to a state in $\mathbf{S_1}$**.** Both $\mathbf{S_1}$ and the set of states that satisfy Φ are represented by BDDs. The transition relation *R*, being a subset of the set of 2N bit strings can also be represented by a BDD. If f_1 is the Boolean function associated with $\mathbf{S_1}$, and χ_Φ is the characteristic function of the set of states satisfying Φ, we can compute the Boolean function associated with $\mathbf{S_2} \supseteq \mathbf{S_1}$, as $f_2(\text{x}) = f_1(\text{x}) \vee (\chi_\Phi(x) \wedge \exists\, x'.(R(x,x') \wedge f_1(\text{x}))$. The key point is that all these operations can be computed using the BDDs associated with each Boolean function.
3. Now we iterate step 2, looking for states that both satisfy Φ and have a transition to a state in $\mathbf{S_2}$.
4. Keep iterating until the BDD associated with $\mathbf{S_j}$ is the same as the one associated with $\mathbf{S_{j+1}}$.
5. Check if the set of initial states is included in $\mathbf{S_j}$. This can be done by computing the AND of the BDD associated with the set of initial states and the one associated with $\mathbf{S_j}$ and checking the resulting BDD is the same as the one for the initial states.

As we can see, the algorithm looks for a mathematical object (a set) that is extremal in the sense that is the larger set that satisfies the property. This interpretation gives us the idea that the problem may be recast as some sort of optimization. This was the idea put forward at the Vanderbilt meeting. The hope was that such an optimization problem may be suitable for implementation with DW2.

4.1.7 Issues with casting computation of extremal BDD as optimization problem

In order to cast this computation of an extremal BDD as a combinatorial optimization problem suitable for implementation with DW2, we need to be able to accomplish two tasks: first, encode BDDs as binary strings (the space over which DW2 performs optimization), and second, construct a cost function that whose minimum is associated with the extremal BDD we are looking for. We will now discuss what we see as major roadblocks for these two requirements.

BBD encoding as binary strings: BDDs are used to encode Boolean functions on N-bit strings. Since these functions are uniquely associated the subsets of N-bit strings, there are 2^{2^N} such functions. However, we can only encode a space of size 2^N on DW2. So here is the first major obstacle to optimizing over BDDs using DW2: we will be able to encode all BDDs only of systems with *log (Nqubits)* states, and since the number of qubits is currently 512 and not expected to grow exponentially, this approach will only work for toy models and will not scale for even moderately sized problems.

A way around this issue could be to restrict the set of BDDs over which the optimization is performed. However, there is no clear guidance on how to choose this subset. Furthermore, such a subset must include the BDD associated with the solution for the computation, but we have no way of knowing if that will hold. If we proceed anyway, we will not have any assurance that the BDD obtained has any of the properties required by the solution. The only way to check this will be to perform an exhaustive testing of such a BDD, but this defeats the purpose of using BDDs in the first place.

Cost function for BDD optimization: even if we assume that we have enough qubits to encode all possible BDDs, we still need to construct a suitable cost function whose minimum is associated with the extremal BDD sought. From the algorithm presented above it is clear that the sequence of BDD generated represent subsets of the state space with increasing size, so the size may be a candidate for a cost function. But it is also clear that there are more constraints that the set needs to satisfy, i.e., the states it contains need to satisfy a certain formula in some temporal logic. The parameters at our disposal when constructing a cost function for DW2 are the local fields and the interaction between qubits. This results in a quadratic function, and it is not clear that this form has the power to encode the required properties of the states. At the very least, this mapping (if possible) would require a lot of preprocessing before it can be implemented in DW2, and this will likely erase any gains produced by running the optimization in the quantum computer.

4.1.8 Results of BDD implementation using DW2

As discussed above, implementing an optimization approach to solve the same fix point problem that is usually addressed with BDDs has several obstacles. It may be the case that these obstacles can be overcome in some cases (for example, if we have enough knowledge to restrict the set of BDDs over which the optimization will take place), but at that point in the research we believed that the cost of focusing on this problem and identifying favorable instances would have been too high for the expected benefit. Furthermore, it would have negatively affected the research on other aspects of model checking that seemed more likely to have a payoff. It was then decided not to pursue this avenue of research as part of the QCHECK project.

4.2 Benchmarking of DW2 on MAX-2-SAT versus MaxWalkSAT

As discussed in previous quarterly reports, a fair comparison of the performance of the chip is obtained when benchmarked against a probabilistic solver like MaxWalkSAT, instead of an exact solver like AK-MAXSAT. An exact solver will typically take a longer time to run since it is not only providing the solution, but also a guarantee of optimality. On the other hand, the D-Wave processor falls under the class of probabilistic solvers, where running an instance will result in an answer that corresponds to the optimal solution with probability $p<1$. Hence, in order

to get a higher confidence in the result, the solver needs to be run many times, and this must be included in the computation of the runtime. The figure of merit will be the number of repetitions needed to find the best answer for an instance with a probability above a certain threshold.

To run this benchmark we generated random instances of MAX-2-SAT that respected the connectivity of the Chimera graph, so that they could be run directly on the D-Wave processor. The instances had N variables and M=2N clauses, where N ranged from 20 to 500 (in steps of 20). For each instance we used an exact solver to determine the optimal solution, so we can use it to compute with what probability the D-Wave processor and MaxWalkSAT find the correct answer. The plot below shows the results we obtained on this benchmark using a 2.5GHz desktop processor to implement MaxWalkSAT.

4.2.1 Analysis of the benchmarking results

The results of Figure 7 show the comparison between the performance of DW2 against the MaxWalkSAT algorithm run on a Mac Pro desktop (2.6 GHz processor). We plotted the logarithm of the runtime (in microseconds) against the number of variables. What we see from this experiment is that the DW2 processor has a better performance for problems above N=40 variables. This is an encouraging result, although it should be fair to point out some caveats.

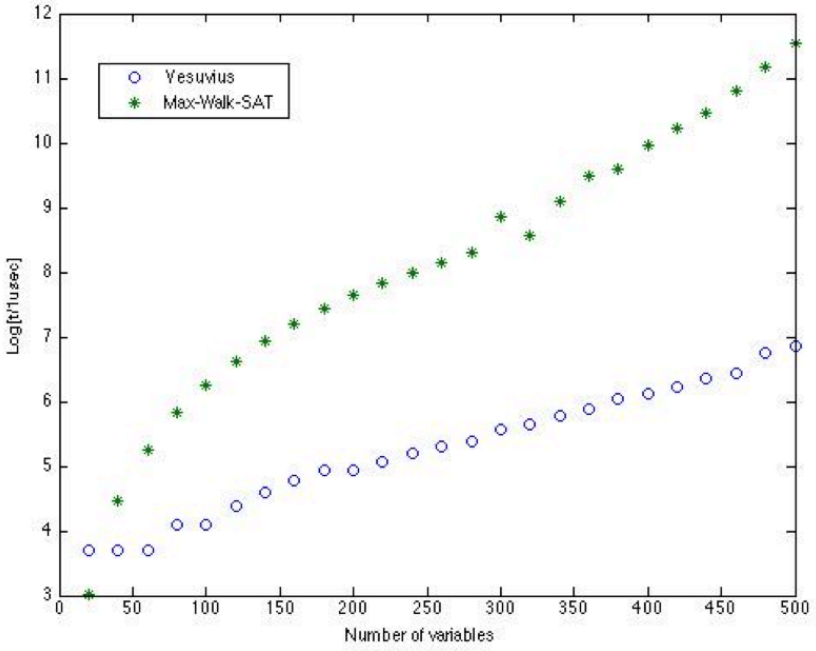

Figure 7. Benchmarking results for MAX-2-SAT

First, the MaxWalkSAT algorithm is designed to tackle general MaxSAT problems and no steps were taken to optimize it for the Chimera connectivity of the problems in the ensemble we tested. In some sense, one can see this issue as providing some kind of advantage to the DW2 processor, since the problems are native to its underlying architecture. This is an issue that should always be kept in sight when interpreting benchmarking results for DW2.

Second, we have only run these instances on a moderately powerful desktop computer. A faster processor will certainly increase the performance of MaxWalkSAT. This is another issue that makes comparing the DW2 quantum device with a classical algorithm somewhat difficult. We do not have any guidance to decide against which classical hardware should the DW2 performance be measured. One could consider the cost of the systems (i.e., look for a classical computing system of similar cost), but this will give the classical approach an unfair advantage, given that classical hardware and algorithms have benefited from decades of research and investment. Quantum systems like DW2 have been around for only a few years and their development is still at the very early stages.

Another issue to point out is that for the DW2 system we are only counting the time required to run the annealing portion of the optimization. The system has a significant time overhead that is required to program the device and to measure the results at the end. On the other hand, this overhead is constant (and being improved in newer designs) and will play less of a factor in future generations of the chip having a larger number of qubits.

4.3 Implementation of a heuristic embedding tool

The heuristic embedding tool was implemented as a Matlab program that communicated with the DW2 processor through a special function call, gathered its output and computed the gradient of the relative entropy, updated the Ising model and submitted it again to the DW2 processor. The full code is submitted as an addendum to this report. We will now discuss the main components of the code.

4.3.1 Code structure

The main code is the Matlab function `SEBREMforQUBO` (SEBREM stands for Sequential Embedding By Relative Entropy Minimization):

- Inputs:
  1. `Qfull`: a matrix encoding the full QUBO problem to solve
  2. `beta`: a parameter that characterizes the temperature of the Gibbs distribution associated with `Qfull` (usually taken to be 1).
  3. `Niterations`: the number of iterations of the sequential embedding.
  4. `EmbeddingFlag`: a flag that determines the initial embedding. It could take three values: `EmbeddingFlag` =1, applies a greedy embedding that tries to map the variables to the qubits in a way that preserves the couplers with the largest absolute value; `EmbeddingFlag = 2`, applies a modified version of the greedy embedding, where when various options are presented when choosing which qubit a variable will be assigned to, it chooses one of them at random (the previous embedding would choose the first qubit in the list); `EmbeddingFlag = 3`, applies a randomized direct embedding, i.e., it randomly assigns variables to qubits.

5. `Step`: controls the size of the step when updating the (*h,J*) parameters in the Ising model.

6. `Display`: (0 or 1) turns off and on the display of results after each iteration.

- Outputs: 1. `RelativeEntropy`: a vector of values of the relative entropy at each iteration step.

  2. `BestSolutions`: a matrix whose columns are the best solutions found at each iteration (a solution is a binary string).

  3. `BestObjective`: a vector with the best value of the objective function found at every iteration.

  4. `BestSolutionFound`: a binary string representing the best solution found over all iterations.

  5. `BestObjectiveFound`: the value of the objective function corresponding to the best solution found.

The code starts by fixing the values of certain parameters and initializing matrices that would store the results generated by the iterative procedure. Then it generates the initial embedding (depending on the value of `EmbeddingFlag`) and then starts the iteration that construct the sequential embedding. Each iteration step has the following structure:

1. Solve the Ising model using the DW2 using the function `IsingConnectSolve`, which is called using the parameters (*h_chimera,J_chimera*) . We multiply the parameters by 0.5 in order to increase the number of samples generated (this is equivalent to raising the temperature).
2. Extract an empirical distribution from the DW2 output, i.e., the solutions sampled, their frequency and their associated energy.
3. Compute the objective function associated with the matrix `Qfull` on all solutions sampled in the previous step.
4. Extract the solutions that minimize the value of the objective function.
5. Compute the relative entropy.
6. Compute the gradient of the relative entropy.
7. Update the values of the Ising model given by (*h_chimera,J_chimera*), using the gradient information. This step also adds another heuristic (implemented inside the function `UpdateIsingModel`): it adds to the Ising model another Ising model that has the Chimera connectivity and the property that the best solution seen so far is its ground state. The idea is to reward direction in the state of parameters that produced good solutions. This heuristic is controlled by the parameter `MIXFACTOR`.
8. Information is displayed (depending on the value of the `Display` flag) and saved.

9. Go back to step 1 until `Niterations` are performed.

4.3.2 Discussion of code performance

The main objective of this task was to develop a tool that would allow us to solve problems with the DW2 processor that would not fit in the underlying architecture given by the Chimera graph. In that capacity, the code generates a sequence of Ising models that respect the Chimera graph, with parameters that are adjusted in such a way that the solutions obtained are better solutions of the original non-Chimera problem. The whole method can be seen as an optimization in the space of parameters (h,J), which is a continuous, bounded subset of $\mathbf{R}^n$ (the space of n-tuples of real numbers). The boundaries are given by the limited range that the parameters can take when programming the device: the local fields lie within -2 and 2, while the couplers take values between -1 and 1. The function that we are trying to optimize in this space is the relative entropy, and although we have an analytical expression for it, the function is not convex and we have no guarantees about finding the absolute minimum.

It is then hard to quantify the performance of the code in general, and we are then limited to study its behavior in examples of interest. The basic characteristic that we would want the code to have, is that it would improve the solutions provided by the initial embedding. The initial embedding is in some sense the best we can do to solve, in one shot, a problem that does not fit the processor's connectivity. Different strategies for this embedding will result in different quality of solutions. We considered three types of initial embeddings: two that tried to capture some of the structure of the problem (greedy embeddings), and one that essentially mapped the problem into the processor in a random way. An interesting feature we found is that, even though the greedy strategies provided better initial solutions than the randomized embedding, by implementing the sequential embedding we were always able to generate better solutions after a few iterations, irrespective of the initial embedding we chose. Given that the greedy embeddings required some non trivial preprocessing (i.e., finding such embedding) that required extra computational resources, we realized that a randomized embedding would provide good solutions without the upfront computational cost.

The time performance of the sequential embedding depends on the size of the problem being considered, mainly because the computation of the gradient can be expensive if care is not taken to code it efficiently. We implemented the computation in Matlab, and we took care of vectorizing the calculations as much as possible in order to avoid any loops (that are notoriously slow in Matlab). For problems of around 500 variables, the gradient computation step would take only a few seconds. We found that in most problems, only a few tens of iterations were enough to reach a point where the solutions would no longer improve (we suspect we were either reaching a local minima of the relative entropy, or that the approximations we made were no longer valid). So in summary, for a 500 variable problem, a run time of a few minutes will be the most we needed to run in order to find the best solutions this method would provide. Applying this method to future generations of the processor, that would support more than a thousand variables, would certainly benefit from speeding up the gradient computation by using faster languages.

The algorithm has several parameters that can be used to try to improve convergence and behavior. In this project we did not have the time required to analyze them in more detail and try

to find their optimal values. The values we used were born out of trial and error, where the driving feature was that the code generated improving solutions in a reasonable amount of time.

In summary, the tool we developed succeeded in allowing us to produce good solutions for optimization problems that did not fit into the processor's connectivity. These solutions were better than the ones obtained using single-shot approximate embeddings. We can always take any approximate embedding as the initial embedding, and we believe that the algorithm implemented here will always improve the solutions initially obtained.

4.4 Integration of DW2 into CEGAR loop

As discussed in previous sections, the central goal of this project was to integrate the capabilities of the DW2 processor into the CEGAR framework of model checking. We identified two key steps in the CEGAR loop where combinatorial optimization problems needed to be solved in order to proceed:

i. Solving a SAT problem to verify the validity of an abstract counterexample
ii. Solving an ILP problem to find the smallest increase of the abstraction in order to get rid of a spurious abstract counterexample

The first of the two tasks, at least in the form we were able to cast it, turned out not to be a good fit for the capabilities of the DW2 processor. The second task provided much more promising results.

4.4.1 Verifying the validity of abstract counterexamples

Given an abstract counterexample, checking its validity consists in verifying if the corresponding trace in the original system also provides a counter example to the property we are trying to prove. This problem can be formally reduced to checking the satisfiability of a Boolean formula that encodes the existence (or non existence) of the required trace in the original system. In order to study this task, we generated abstract problems in New Symbolic Model Checker for System Verification (NuSMV), we checked the abstract models, and reformulated the counterexample checking problem in Linear Temporal Logic (LTL). We wrote a program that read abstract counter-examples, and generated from them LTL formulas that would be satisfiable if and only if the abstract counterexamples were valid, i.e., they were associated with actual counterexamples in the original system. Then we exported the resulting SAT problem using a canonical format, converted them into QUBO problems and tested on DW2.

Our first experiments were with the problem of testing abstract counter-examples for soundness. We then translated these LTL formulas, and the system models, into satisfiability problems, using a standard format known as DIMACS. These SAT problems were then be reformulated and submitted to the DW2 to be solved.

We worked with the NuSMV system developed by Fondazione Bruno Kessler (FBK), CMU, and the Universities of Trento and Genova (http://nusmv.fbk.eu). NuSMV is the next generation, open source version of the original Symbolic Model Checker for System Verification (SMV) model-checking tool. SMV was the first BDD-based model-checker, developed by McMillan at CMU. NuSMV extends the original SMV, and offers both BDD-based model-checking and SAT-based bounded model-checking. NuSMV provided the model-checking of the abstract

models, and translated our LTL claims into SAT problems (NuSMV provides SAT problem translation to support external SAT tools for BMC).

For our experiments with soundness checking, we manually abstracted various models from the NuSMV distribution. These models included some properties to check, and we added other properties of our own. We added some of our own properties since most of the properties in the distribution were properties that the models satisfied, so we needed to have additional cases covering unsafe models.

To generate the abstract counter-examples, we used NuSMV's BDD-based model-checking on the manually abstracted models. NuSMV can be configured to write counter-examples in an Extended Markup Language (XML) form that is very easy to parse. We are grateful to FBK personnel for helping us understand the XML format, and for responding to our issues with the format.

Reformulation: The key to our checking process is reformulating the validity checking problem as an LTL claim, since this enables automatic generation of the SAT problem. The abstract counter-example is a sequence of abstract states, as_i, each of which is an assignment of values to propositional variables. Since these are abstract states, the assignments are to only a subset of the propositional variables of the full model. Again, since this is an abstraction, a single abstract transition may correspond to multiple concrete transitions. So for an abstract counter-example $as_0, as_1 \ldots as_n$ to be valid, there must be a valid sequence of concrete states $s_0 \ldots s_i \ldots s_m$ such that $as_0 \subseteq s_0$, $as_1 \subseteq s_{i,\ldots}$ $as_n \subseteq s_m$. Note that the subset relation is equivalent to saying that the concrete state entails the abstract state. We may formulate this in LTL as follows:

$$as_0\, X(as_1\, X(as_{2\ldots} X\, as_n))) \text{ (Property 1)} \qquad (24)$$

That is, we begin in a state satisfying as_0, eventually we reach a state satisfying as_1, then eventually we reach a state satisfying as_2, and so on, until we reach a state satisfying as_n, all in the context of the full, original, concrete model.

The above query cannot be used as written to check validity, however. As an LTL property, it is effectively claiming that in *all* runs of the model, we start in a state satisfying as_0, reach a state satisfying as_1, etc. To use this formulation, we must *invert* it, and charge the solver to prove to us that it is *impossible* to satisfy Property 1. If the negation of Property 1 is valid, then the abstract counter-example is *invalid:*

$$\neg as_0\, X(as_1\, X(as_{2\ldots} X\, as_n))) \text{ (Property 2)} \qquad (25)$$

Implementation: To support these experiments, we wrote a program that parsed the XML formatted counterexamples, and generated from them LTL claims in the form of Property 2, expressed in NuSMV's input language. These LTL claims, together with the original (concrete) system model, were then submitted to NuSMV, and bounded model checking SAT models in DIMACS notation were extracted. These DIMACS problems were then translated into problems for the D-Wave system.

Unfortunately, experience has shown that the D-Wave quantum adiabatic optimization process is not a good choice for this kind of decision problem. The D-wave is well suited to approximate optimization problems, where best effort is what matters. But in the case of decision problems, a best effort that misses is not a useful approximation; it is simply wrong. The CEGAR algorithm shows that approximate solutions that can be wrong are sometimes acceptable, if the

approximations are *safe* (conservative), and if there is a means to refine the computation. So it is possible that a revision of our approach here, where the D-Wave would only give erroneous results that mistakenly concluded that the counter-example was *valid*, and never give a false invalid result, would be useful. Such a technique would never cause us to mistakenly certify a system as safe when it wasn't, and would never send us on a wild-goose chase to refine an already sound counter-example.

4.4.2 Refining the model

The second part of the QCHECK project involved using the D-wave to solve the optimization problems arising in the refinement part of the CEGAR algorithm. Recall that this involves finding an optimal refinement of the model. In the specific version of CEGAR that we used, this was, specifically, finding a minimal set of propositional variables, previously abstracted away, to add to the abstract model. This is formulated as an ILP, choosing a set of variables that are sufficient to create a new abstraction in which the dead end states and the bad states of an invalid counter-example, are guaranteed to be in different abstract states.

Our work in this area was more successful than our work on checking abstract counter-examples. This was primarily due to the fact that the abstraction-refinement is a better fit to the D-wave's capabilities. In particular, this problem is an optimization problem, rather than a decision problem, and if the solution is less than perfectly optimal, it simply costs us more work in the model-checking phases of the CEGAR process: a suboptimal answer can *never* lead to an unsound conclusion.

Our original plan was to find an off-the-shelf model-checker with an implementation of the CEGAR algorithm, from which we could extract ILPs or, as a second choice, a model checker in which we could perform steps of the algorithm and from which we could extract partial results. Unfortunately, our attempts to find such a tool were not successful. We contacted the authors of the original CEGAR paper, and while they offered a number of very helpful suggestions, the code for the original implementation had been lost. We consulted FBK about NuSMV, and although its successor, NuXmv, will contain an implementation of the CEGAR algorithm, it was not in a condition for release to us.[1] We also investigated whether it would be possible to work the CEGAR algorithm "around" a tool, in the way we were able to do our counter-example checking "around" the NuSMV tool. We concluded that this would not be possible, since the data structures needed to identify the dead end and bad states were not visible to the user of NuSMV. We also investigated whether it would be possible to implement the CEGAR algorithm by scripting NuSMV. This technique seemed very promising: the NuSMV code is well structured, and we were able to construct NuSMV data structures and exercise some of its functions through a foreign function interface from the Common Lisp programming language. Unfortunately, we concluded that for this small program, we didn't have the resources to dig deeply into the NuSMV sources. Scripting NuSMV would be a promising direction to take given a larger-scale program.

[1] It has since been released, but so far only in closed-source form, so would not be suitable for our purposes.

After we had to abandon the plan of using an off-the-shelf tool, our next step was to manually go through the steps of the CEGAR algorithm, using two BDD "desk calculators," to check the abstractions, and to identify dead ends and bad states. In particular, we used both the iben[2] and bddc[3] tools. We learned a number of valuable lessons about the structure of models that would give rise to valid and invalid abstract counter-examples, and designed some small problems. After developing a small number of small problems, we were unable to make larger problems: the need to work directly with propositional logic was too burdensome and inefficient.

At this point we were nearing the end of the program, and realized that we needed our own implementation of the CEGAR algorithm, so that we could run examples more easily. We were fortunate to find that the CU Decision Diagram (CUDD) open source Ordered Reduced Binary Decision Diagram (OBDD) library,[4] developed by Fabio Somenzi at Colorado University, offers a Perl application programming interface (API). Using this API we could rapidly build our own implementation around the BDD operations, developing interactively.

The CUDD library provided one of the pieces of the solution: it remained to make or choose an input language. We considered the SMV language, but concluded that writing a parser would require too much effort. We had consulted Ofer Strichman, one of the authors of the original CEGAR paper, and he suggested that we use the AIGER – And Inverter Graph – notation.[5] The AIGER format offered a number of advantages: (1) it is very easy to parse; (2) it is used for hardware model-checking competitions, so there are a large number of preexisting models available; (3) it maps nicely to the abstraction framework of CEGAR. With respect to point (3), in addition to AND gates and Inverters, AIGER models contain latches. A CEGAR-compatible abstraction technique is to abstract away latches by treating them as if they are inputs. Finally, point (4), there are existing BMC tools for AIGER models that we could (and did) use to check our results when building and debugging our CEGAR implementation.

In practice, the AIGER notation did not provide a perfect input solution: although it is higher level than the propositional logic that we used when working with the BDD calculators, it was still too low level. In particular, AIGER offered no way to capture higher level building blocks such as, for example OR-gates, XOR-gates, half-adders, etc. In order to build models that would offer interesting ILPs, we needed to be able to reuse model components. To support such reuse, we added a simple macro-language, `laig` (for Lisp-flavored AIG), and wrote a simple macro-expansion facility. We discuss this in more detail below.

Three other issues remained. First, although the `laig` notation made it possible to reuse model components, allowing us to make bigger and more interesting circuits, it did nothing to help formulate the properties we needed to check. In our successful test cases, we were checking properties of the form "in a valid trace it is impossible to reach a state satisfying *P*." The proviso "in a valid trace" was necessary in order to capture bounding assumptions such as "there are no

---

[2] http://sourceforge.net/projects/iben/

[3] http://www-verimag.imag.fr/PEOPLE/Pascal.Raymond/tools/bddc-manual/bddc-manual-pages.html

[4] http://vlsi.colorado.edu/~fabio/CUDD/

[5] http://fmv.jku.at/aiger/

more than two sensor failures" in our example avionics model. In practice, assumptions like "valid trace" unpacked into temporal properties which were difficult to formulate in AIGER and which `laig` notation did not support as well as possible. To capture temporal properties in AIGER notation involved adding latches that captured property components so that temporal assertions could be evaluated. While `laig` allowed us to capture repeated components of such translations, an automatic translation of temporal properties would have been helpful. Another problem was that the existing AIGER models were not very well documented, so they didn't provide all the CEGAR examples we had wanted. Nevertheless, the availability of examples, with gold standard computation results (from the BMC checker), was immensely useful in development and debugging. Finally, the abstraction relationship of treating latches as inputs turned out to be more subtle than we had originally anticipated: different treatments of abstraction in model-checking used subtly different definitions, so that getting consistent results required substantial debugging and rework.

4.4.3 CEGAR Implementation

To summarize the implementation, it contained the following components:

Input:

- The Lisp-style AIGER language (`laig`) and translator. Input models are formulated as `laig` files, and translated into
- Single-output AIGER models. Single-output models are used to capture reachability checks in AIGER models. A model formulated in this way is safe if it can never output 1.
- A single-output AIGER model is read by our AIGER parser, and translated into internal data structures.

CEGAR Process:

- An initial abstraction is computed, which hides all of the latches that are not direct components of the single AIGER output.
- LOOP
  - Our solver checks reachability of the unsafe state in the abstract model. This is done using a search, implemented as a Perl loop (based upon code from Ed Clarke's model-checking text), whose primitive computations are done using the CUDD BDD library.
  - If the unsafe state is *not* reachable, we return "safe" and exit.
  - If the unsafe state *is* reachable, we compute an abstract counterexample.
  - We check the abstract counterexample for validity. This is done according to the CEGAR algorithm, using a loop over the images computed in the reachability search.
  - If the abstract counterexample is valid, we return "unsafe" and exit.

- If the abstract counterexample is *invalid,* we identify the dead end and bad states. We compute an ILP, each of whose rows captures the difference between one dead state, *d,* and one bad state, *b.* The columns of the ILP are latches that are hidden, and each row assigns 1 to propositional variables whose values differ in *b* and *d,* and zero to all other entries.
- The CEGAR algorithm publishes the ILP to the D-wave solver, and reads a solution to the ILP. The solution to the ILP specifies a set of latches that should no longer be hidden.
- We compute a new abstract model, by "unhiding" the set of latches indicated by the D-Wave solver and go to LOOP.

We were pleased to find that it was very easy to interface the CEGAR loop with the existing D-Wave solver. The output format was very easy to print and to parse, as was the format of the answers (a list of latch indices). This holds promise for further work on the CEGAR algorithm, and for other applications in which an outer loop is to be wrapped around the D-Wave solver.

4.4.4 ILP Problems

Particular structures in model-checking problems pose interesting challenges for abstraction refinement. Or, put differently, many spurious counter-examples can easily be eliminated by adding one or two new latches. More challenging problems arise in the presence of two features. First, there must be multiple different concrete paths that correspond to a single spurious abstract counterexample. This arises when there are functions in the counterexample that involve a wide variety of variables in a non-trivial Boolean function. The second condition is that these Boolean functions must not include intermediate values that are computed and latched. If there is such structure, then the CEGAR algorithm will simply identify whether or not the set of values so latched are consistent and either confirm the abstract counter-example or reject it, without need for complex reasoning (this is the strength of the CEGAR algorithm). We conjecture that the challenging cases for CEGAR refinement are characteristic of models of complex circuits. They may be less likely to arise in models of software, where storage of intermediate results is more common. As the D-Wave system gets more powerful, and capable of handling larger and larger models, we believe that we will see more non-trivial ILPs in CEGAR refinement.

4.5 Implementation of CEGAR based model checking example

We describe here an example run of the CEGAR algorithm with the D-Wave solver providing answers to the ILPs.[6] The test model[7] was inspired by an avionics example problem (the

[6] The CEGAR program can be run without the D-Wave in the loop, simply picking a single implicated variable to "unhide" at each iteration. Of course, this can perform arbitrarily badly; it is only of use on small test runs, to debug other parts of the CEGAR loop.

Rockwell Flight Control System (FCS) 5000 problem [17]), but does not correspond to any real hardware or software. The property checked is a mutual exclusion property: there are two control modes, "left" and "right," and the invariant is that the system should never have both control modes active simultaneously.

4.5.1 Verification summary

Initially, the model is almost completely abstract. Recall that the property to be checked is of the form "in a valid trace, the left and right control modes cannot both be engaged." The only latch involved in this claim is the "trace valid latch." This allows for a trivial counterexample, since "left control mode on" and "right control mode on," are essentially treated as inputs. This is an invalid counterexample, for reasons having to do with system function -- there is substantial logic behind whether or not the control modes are engaged – and the logic of the property -- there is substantial logic in the memory of tracking whether a given trace is valid or not. The first several refinements do not require any optimization: they are obvious refinements that gradually add logic for the full temporal chain needed to decide trace validity.

The first interesting ILP arises when the model has been elaborated around the temporal logic of deciding trace validity. There is now a spurious counterexample to the effect of "turn the right controller on, turn the left controller on, wait for three cycles, then turn the valid latch on." The model actually enforces that the control modes can only be engaged if sensors detect the right condition and enough of the redundant sensors vote to enable it. This meets the criterion above for an interesting ILP: there are many combinations of the sensor latches that could lead to a successful vote to engage the sensor modes, and the ILP solver must choose a minimal covering set.

The second, and last, interesting ILPs arise from a spurious counterexample in which the sensor inputs are arranged in such a way that both the left and right control modes can be simultaneously engaged. This is a spurious counterexample because of the trace validity checking: it turns out that such a constellation of sensor inputs cannot happen in valid traces. Note, however, that we don't simply eliminate *all* inconsistent sensor value combinations: we simply stipulate a limit on the number of sensors that can simultaneously give the wrong detections. The trace validity constraints capture the notion that, e.g., the aircraft cannot be taking off and landing at the same time. In solving the final interesting ILP, the solver identifies a set of variables that is sufficient to capture enough of the validity-checking logic to eliminate these inconsistent sensor combinations.

This summary has skipped over the refinement steps that do not require the intervention of the ILP solver. There are many such steps, where identifying a set of latches to add to the model is trivial. This arises when, e.g., the same latch appears in all rows of the ILP. In those cases, the CEGAR loop does not invoke the D-Wave. We give the full transcript below.

4.5.2 Detailed transcript

Initially, the model is almost completely abstract. The only un-hidden variable is the one corresponding to "the trace is valid" (literal 52). Unsurprisingly, this leads to a trivial

---

[7] We give the `laig` source as an appendix.

counterexample: we just turn on the right and left controllers (since they are unconstrained in this abstract model).

This abstraction is too simple, for two different reasons: The first is the interesting reason, which is that the counterexample must take into account the surrounding environment, as redundantly sensed. The less interesting reason is that there is a chain of latches for the on/off state of the left and right controllers; this is necessary to synchronize the state of the system with the state of the part of the circuit that tracks validity of the trace. A trace that's invalid (e.g., we are both at and not at an airport, simultaneously) must be weeded out.

To be precise, we are verifying that we never enter the bad state of "left3 is on," "right3 is on," and "the trace is valid."

The first several steps of the CEGAR refinement process introduce, one by one, the latches in the three-step delay chains from "right1" to "right3" and "left1" to "left3."

```
Made new Aiger model with 9 inputs, 40 latches, 1 outputs, and 42
and gates.
Reading 40 latches.
Reading 1 outputs.
Reading 42 and gates.
Done reading file and building model.
Checking abstract model, hidden literals are: 20 22 24 26 28 30 32
34 36 38 40 42 44 46 48 50 54 56 58 60 62 64 66 68 70 72 74 76 78 80
82 84 86 88 90 92 94
Making transition relation
Done making transition relation
Preparing for model-checking.
Starting fixed point computation.
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Goal reached
Goal Reached in 2 steps
Counterexample:
0
      Inputs: 111111111
      Latches: 0000000000000000000000000000000000000000
      Output: 0
1
      Inputs: 111111111
      Latches: 1111111111111111110111111111111111111100
      Output: 0
2
      Inputs: 111111111
      Latches: 1111111111111111111111111111111111111111
```

```
      Output: 1
Abstract counterexample:
      000
      100
      111
Abstract counterexample is invalid at step 2.
ILP is:
92 94
Splitting on literal 92
```

The first refinement is to add one of the two latches (literals 92 & 94), representing the state of the left and right controllers. In this case, the algorithm adds the literal 92 (“left3”). This does not require the intervention of the D-Wave solver; the ILP is trivial (the format “92 94” represents a linear constraint in which the sum of literals 92 and 94 must be greater or equal than 1).

```
Checking for goal
Goal reached
Goal Reached in 3 steps
Counterexample:
0
      Inputs: 111111111
      Latches: 000000000000000000000000000000000000000000
      Output: 0
1
      Inputs: 111111111
      Latches: 111111111111111111011111111111111110100
      Output: 0
2
      Inputs: 111111111
      Latches: 111111111111111111011111111111111111101
      Output: 0
3
      Inputs: 111111111
      Latches: 111111111111111111111111111111111111111
      Output: 1
Abstract counterexample:
      0000
      1000
      1101
      1111
Abstract counterexample is invalid at step 2.
ILP is:
88 94
Splitting on literal 88
```

The next counterexample is almost as trivial. The solver realizes that the left channel cannot simply be turned on instantaneously, but still generates a very quick path to failure (only three steps long), taking into account a delay in turning the left channel on. Note that this delay comes

partly because there is a delay from the logic of checking the switch and the state-sensing logic, but also because the model has a three-step delay to compute the validity of the trace.

This counterexample is shown to be spurious: the abstract model now takes into account the delay in turning on the left controller, but not yet the right controller. The culprit for this three-step counterexample is that there must be a progression (at least) through latches “right2” (88) and “right3” (94). The solver has another trivial refinement problem, to choose between 88 and 94, which it does by choosing in numerical order (again, no need for the ILP solver; there is only one row to the ILP).

```
Starting fixed point computation.
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Goal reached
Goal Reached in 4 steps
Counterexample:
0
      Inputs: 111111111
      Latches: 0000000000000000000000000000000000000000
      Output: 0
1
      Inputs: 111111111
      Latches: 11111111111111111101111111111111111010100
      Output: 0
2
      Inputs: 111111111
      Latches: 11111111111111111101111111111111111110101
      Output: 0
3
      Inputs: 111111111
      Latches: 11111111111111111101111111111111111111101
      Output: 0
4
      Inputs: 111111111
      Latches: 11111111111111111111111111111111111111111
      Output: 1
Abstract counterexample:
      00000
```

```
      10000
      11001
      11101
      11111
Abstract counterexample is invalid at step 2.
ILP is:
82 94
Splitting on literal 82
Checking abstract model, hidden literals are: 20 22 24 26 28 30 32
34 36 38 40 42 44 46 48 50 54 56 58 60 62 64 66 68 70 72 74 76 78 80
84 86 90 94
Making transition relation
Done making transition relation
Preparing for model-checking.
Starting fixed point computation.
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Goal reached
Goal Reached in 5 steps
Counterexample:
0
      Inputs: 111111111
      Latches: 000000000000000000000000000000000000000
      Output: 0
1
      Inputs: 111111111
      Latches: 111111111111111111011111111111011010100
      Output: 0
2
      Inputs: 111111111
      Latches: 111111111111111111011111111111111010101
      Output: 0
3
      Inputs: 111111111
      Latches: 111111111111111111011111111111111110101
      Output: 0
```

```
4
      Inputs: 111111111
      Latches: 111111111111111111011111111111111111101
      Output: 0
5
      Inputs: 111111111
      Latches: 111111111111111111111111111111111111111
      Output: 1
Abstract counterexample:
      000000
      100000
      110001
      111001
      111101
      111111
Abstract counterexample is invalid at step 2.
ILP is:
94
Splitting on literal 94
Checking abstract model, hidden literals are: 20 22 24 26 28 30 32
34 36 38 40 42 44 46 48 50 54 56 58 60 62 64 66 68 70 72 74 76 78 80
84 86 90
Making transition relation
Done making transition relation
Preparing for model-checking.
Starting fixed point computation.
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Goal reached
Goal Reached in 5 steps
Counterexample:
0
      Inputs: 111111111
      Latches: 000000000000000000000000000000000000000
      Output: 0
1
```

```
      Inputs: 111111111
      Latches: 11111111111111111101111111111111011010000
      Output: 0
2
      Inputs: 111111111
      Latches: 11111111111111111101111111111111111010100
      Output: 0
3
      Inputs: 111111111
      Latches: 11111111111111111101111111111111111110101
      Output: 0
4
      Inputs: 111111111
      Latches: 11111111111111111101111111111111111111101
      Output: 0
5
      Inputs: 111111111
      Latches: 11111111111111111111111111111111111111111
      Output: 1
Abstract counterexample:
      0000000
      1000000
      1100100
      1110101
      1111101
      1111111
Abstract counterexample is invalid at step 2.
ILP is:
90
Splitting on literal 90
Checking abstract model, hidden literals are: 20 22 24 26 28 30 32
34 36 38 40 42 44 46 48 50 54 56 58 60 62 64 66 68 70 72 74 76 78 80
84 86
Making transition relation
Done making transition relation
Preparing for model-checking.
Starting fixed point computation.
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
```

```
Computing successors
Checking for new state
Checking for goal
Goal reached
Goal Reached in 5 steps
Counterexample:
0
      Inputs: 111111111
      Latches: 000000000000000000000000000000000000000000
      Output: 0
1
      Inputs: 111111111
      Latches: 111111111111111111011111111111011000000
      Output: 0
2
      Inputs: 111111111
      Latches: 111111111111111111011111111111111010000
      Output: 0
3
      Inputs: 111111111
      Latches: 111111111111111111011111111111111110100
      Output: 0
4
      Inputs: 111111111
      Latches: 111111111111111111011111111111111111101
      Output: 0
5
      Inputs: 111111111
      Latches: 111111111111111111111111111111111111111
      Output: 1
Abstract counterexample:
      00000000
      10000000
      11010000
      11110100
      11111101
      11111111
Abstract counterexample is invalid at step 3.
ILP is:
84
Splitting on literal 84
```

This refinement process continues, the counterexample still being trivial, and the refinement introducing more of the latches in the three-latch left and right chains.

Quick summary: these refinements eliminate the “just turn the right on and the left on” abstract counterexample. It takes multiple refinement steps to introduce all of the latches in the “right1” … “right3” and “left1”…”left3” delay chains.

```
Goal Reached in 5 steps
Counterexample:
0
      Inputs: 111111111
      Latches: 000000000000000000000000000000000000000000
      Output: 0
1
      Inputs: 111111111
      Latches: 111111111111111111011111111111011000000
      Output: 0
2
      Inputs: 111111111
      Latches: 111111111111111111011111111111111010000
      Output: 0
3
      Inputs: 111111111
      Latches: 111111111111111111011111111111111110100
      Output: 0
4
      Inputs: 111111111
      Latches: 111111111111111111011111111111111111101
      Output: 0
5
      Inputs: 111111111
      Latches: 111111111111111111111111111111111111111
      Output: 1
Abstract counterexample:
      000000000
      101000000
      111010000
      111110100
      111111101
      111111111
Abstract counterexample is invalid at step 2.
ILP is:
70 72 74
70 72 76
70 74 76
72 74 76
70 72 78
70 74 78
72 74 78
70 76 78
72 76 78
74 76 78
No common element in ILP.  Enter literals to split on.
Calling print_ilp
----------------------------------------
Minimized ILP:
70 72 74
```

```
70 72 76
70 74 76
72 74 76
70 72 78
70 74 78
72 74 78
70 76 78
72 76 78
74 76 78
----------------------------------------

Returned from print_ilp
70 72 74
```

At this point, we have an abstract counterexample that effectively says "turn the right controller on, turn the left controller on, wait for three cycles, then turn the valid latch on."[8] Now we get our first interesting ILP, as the solver generates a spurious counterexample, the sensing logic enabling the left controller to turn on is still abstract. In fact, the controller can only be turned on if the sensors detect the right condition and enough of the redundant sensors vote to enable it. There are a large number of combinations of possible sensor result latches that could lead to this voting outcome, and the D-Wave ILP solver chooses a minimal covering set (literals 70, 72, 74).

```
Goal reached
Goal Reached in 6 steps
Counterexample:
0
      Inputs: 111111111
      Latches: 000000000000000000000000000000000000000
      Output: 0
1
      Inputs: 111111111
      Latches: 111111111111111111011111100011011000000
      Output: 0
2
      Inputs: 111111111
      Latches: 111111111111111111011111111111101010000
      Output: 0
3
      Inputs: 111111111
      Latches: 111111111111111111011111111111111100100
      Output: 0
4
      Inputs: 111111111
      Latches: 111111111111111111011111111111111111001
```

[8] Since the logic behind computing validity is still abstract, the system considers that it could just turn the valid latch on.

```
      Output: 0
5
      Inputs: 111111111
      Latches: 11111111111111111101111111111111111110
      Output: 0
6
      Inputs: 111111111
      Latches: 11111111111111111111111111111111111111
      Output: 1
Abstract counterexample:
      000000000000
      100001000000
      111110010000
      111111100100
      111111111001
      111111111110
      111111111111
Abstract counterexample is invalid at step 4.
ILP is:
36 38 40 56
40 56
36 38 56
56
Splitting on literal 56
Checking abstract model, hidden literals are: 20 22 24 26 28 30 32
34 36 38 40 42 44 46 48 50 54 58 60 62 64 66 68 76 78 80 86
Making transition relation
Done making transition relation
Preparing for model-checking.
Starting fixed point computation.
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Goal reached
```

```
Goal Reached in 6 steps
Counterexample:
0
      Inputs: 111111111
      Latches: 000000000000000000000000000000000000000000
      Output: 0
1
      Inputs: 111111111
      Latches: 111111111111111110011111100011101100000 0
      Output: 0
2
      Inputs: 111111111
      Latches: 111111111111111110011111111111111101010000
      Output: 0
3
      Inputs: 111111111
      Latches: 111111111111111110011111111111111111100100
      Output: 0
4
      Inputs: 111111111
      Latches: 111111111111111110011111111111111111111001
      Output: 0
5
      Inputs: 111111111
      Latches: 111111111111111111011111111111111111111110
      Output: 0
6
      Inputs: 111111111
      Latches: 111111111111111111111111111111111111111111
      Output: 1
Abstract counterexample:
      0000000000000
      1000001000000
      1011110010000
      1011111100100
      1011111111001
      1011111111110
      1111111111111
Abstract counterexample is invalid at step 3.
ILP is:
36 38 40 54
40 54
36 38 54
54
Splitting on literal 54
```

The next counterexample has the left sensing on, and a concurrent enabling of the right channel. The model-checker detects that this abstract counterexample is incorrect, because of the trace validity constraints that capture the mutual exclusion between the right and left controller

enabling conditions. In order to check this correctly, the model must be refined to take into account a combination of different bits of validity logic, including the one-shot logic that maintains the validity of the trace,[9] and the left integrity constraints. The next two steps in this refinement are trivial: we add the one-shot validity logic to the model.

This step adds the second latch required for the one-shot validity logic.

```
Goal Reached in 6 steps
Counterexample:
0
      Inputs: 111111111
      Latches: 000000000000000000000000000000000000000
      Output: 0
1
      Inputs: 111111111
      Latches: 111111111111110010011111100011011000000
      Output: 0
2
      Inputs: 111111111
      Latches: 111111111111110010011111111111111010000
      Output: 0
3
      Inputs: 111111111
      Latches: 111111111111110010011111111111111100100
      Output: 0
4
      Inputs: 111111111
      Latches: 111111111111111100111111111111111111001
      Output: 0
5
      Inputs: 111111111
      Latches: 111111111111111111011111111111111111110
      Output: 0
6
      Inputs: 111111111
      Latches: 111111111111111111111111111111111111111
      Output: 1
Abstract counterexample:
      00000000000000
      10000001000000
      10011110010000
      10011111100100
      10011111111001
      11011111111110
      11111111111111
Abstract counterexample is invalid at step 2.
ILP is:
```

[9] If an invalid state is detected in the trace, then it is always invalid.

```
36 38 40 48
36 38 42 48
36 38 48
40 44 48
40 48
42 44 48
44 48
42 48
48
36 38 40 50
36 38 40
36 38 42 50
36 38 50
36 38 42
40 44 50
40 50
40 44
42 44 50
44 50
42 50
50
42 44
No common element in ILP.  Enter literals to split on.
Calling print_ilp
----------------------------------------
Minimized ILP:
48
36 38 40
36 38 42
40 44
50
42 44
----------------------------------------

Returned from print_ilp
36 38 44 48 50
```

At this point, the model has been refined to include enough variables to recognize that the validity flag starts on, and if turned off cannot be restored. This eliminates spurious counterexamples where the adversary takes the system through invalid transitions to turn on the right and left controllers simultaneously, and then flips the valid flag on from off. The new abstract counterexamples involve assignments to the sensor inputs that would allow the aircraft to enable both right and left controllers simultaneously. There are no *valid* assignments that do so, and the model checker detects that the counterexamples are spurious. The final ILP requires the solver to identify a covering set of variables that is sufficient to capture enough of the validity-checking logic to eliminate all of these abstract counterexamples. It chooses enough of the variables to enforce a valid assignment of sensor variables to the left side logic, and enough to enforce the consistency relations between the left and right sides.

```
Goal reached
Goal Reached in 6 steps
Counterexample:
0
      Inputs: 110111111
      Latches: 000000000000000000000000000000000000000000
      Output: 0
1
      Inputs: 110111111
      Latches: 111111110010000110011111100011101100000
      Output: 0
2
      Inputs: 110111111
      Latches: 111111110010000110011111100111110101000
      Output: 0
3
      Inputs: 111111101
      Latches: 111111110010000110011111100111111110010
      Output: 0
4
      Inputs: 111111101
      Latches: 111111111111111010011111100111111111001
      Output: 0
5
      Inputs: 111111101
      Latches: 111111111111111011011111111111111111110
      Output: 0
6
      Inputs: 111111111
      Latches: 111111111111111011111111111111111111111
      Output: 1
Abstract counterexample:
      0000000000000000000
      0000110000001000000
      0000110000110010000
      0000110000111100100
      1111010000111111001
      1111011011111111110
      1111011111111111111
Abstract counterexample is invalid at step 2.
ILP is:
40
42
No common element in ILP.  Enter literals to split on.
Calling print_ilp
----------------------------------------
Minimized ILP:
40
42
----------------------------------------
```

```
Returned from print_ilp
40 42
Will split on:
40 42
OK? y
Splitting on 40 42
Checking abstract model, hidden literals are: 20 22 24 26 28 30 32
34 46 58 60 62 64 66 68 76 78 80 86
Making transition relation
Done making transition relation
Preparing for model-checking.
Starting fixed point computation.
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Checking for goal
Computing successors
Checking for new state
Reached fixpoint after 7 steps
Goal not reached
No abstract path to goal state: circuit is safe.
```

This is *almost* enough to eliminate all of the spurious counterexamples. The final refinement requires the addition of two more variables (40 and 42), which round out some of the input consistency constraints, and then the solver establishes that the failure state is unreachable, and the process concludes.

4.6 Evidence for quantum behavior in the DW2 processor

Even though it was not an explicit part of the QCHECK proposed research, we would like to present very important results regarding the quantum nature of the D-Wave processor that were obtained through research conducted in parallel to the main QCHECK work.

The D-Wave processor was designed to exploit quantum mechanical effects in solving combinatorial optimization problems. However, in order to make the design scalable, certain compromises in the hardware had to be made. The main result of these was that only a restricted set of measurements were allowed to investigate the state of the processor, and hence we lack an important set of tools to determine if the state of the system indeed behaves quantum mechanically. We worked on developing two approaches to address this issue: in one, we designed a very simple problem whose output statistics were different for a simple classical model and a quantum model; in the other, using partial state information about the state of the processor during the annealing we proved, using a novel analytical technique developed for this purpose, that the state of the system has entanglement at some points during the annealing. We give a brief description of both approaches.

4.6.1 Quantum signature

We constructed a simple toy problem involving 8 qubits that had the following characteristics: (i) the ground state was 17^{th}-fold degenerate; (ii) the ground states were divided in two sets, a cluster whose 16 states could be transformed to another ground state by flipping a single qubit, and another single state that required flipping 4 qubits in order to obtain another ground state; (iii) the energy landscape of the problem had no local minima. We studied the evolution of the populations of the 17 ground states according to a simulated annealing evolution (which is a simple classical model of a thermalizing system) and found that the isolated state initially had a higher population than the states in the cluster. On the other hand, a quantum mechanical model of the system predicted that the isolated state would have a suppressed population with respect to the cluster states. We then performed the experiment on the Rainier processor (a 128-qubit processor, that preceded DW2), and found that the experimental statistics agreed with the quantum model and disagreed with the classical model. These results were published in *Nature Communications* [18] (see appendix A).

4.6.2 Evidence of entanglement

The second approach was a collaboration with D-Wave to analyze the entanglement of a set of 8 qubits in the DW2 processor (Vesuvius chip). DW2 has extra controls that allow for certain measurements that were not available for the Rainier processor (DW1). Using these extra controls, an experiment was designed in which one qubit was used as a probe to measure some elements of the state of a set of eight interacting qubits. The extra controls allowed for independent annealing of the probe and the system under study. With this setup, researchers from D-Wave measured the populations of the ground state and the first excited state of the 8-qubit system at different points during the annealing. These two quantities are just 2 of the 65535 real parameters needed to completely determine the quantum state of the system. We at Information Sciences Institute (ISI) developed a new technique to analyze this information that allowed us to conclude that no matter what the values of the unknown parameters were, any state that was compatible with the measured values for the two populations had to be entangled. This was the first experimental proof of the presence of entanglement in the D-Wave processor, and it provides definite evidence of the quantum mechanical nature of the device. This work was accepted for publication in Physical Review X, but has not been published at the time this report is being written. A preprint [19] is attached in appendix A.

5 CONCLUSIONS

The goal of the QCHECK project was to study the feasibility of integrating the capabilities of the D-Wave adiabatic quantum annealer into a model checking framework for the certification of complex systems. The results of this project show that this is indeed possible. By identifying a particular approach to model checking that required solving combinatorial optimization problems, we were able to off load these hard computational tasks to the DW2 processor, construct a hybrid algorithm that applied classical model checking techniques, and call upon the DW2 processor whenever needed. The CEGAR approach to model checking was a very good fit to exploit the strengths of the DW2 device. In fact, the main difficulty encountered in this part of the project, was the lack of a model checking package that was open enough to allow us to extract the required combinatorial optimization problems that needed to be solved by DW2. These packages typically use their own solvers, and do not provide the user with the low level access to extract them. But looking into the future, this project shows that the DW2 processor can be trivially integrated into the CEGAR approach, if the necessary access is provided. Furthermore, any advances in the model checking side and DW2 side will not alter the basics of this integration, so improvements and new developments in both components will only improve the integrated approach.

One of the main reasons for our interest in combining the DW2 device with a model checking approach is the possibility that computational speedups may be provided by the adiabatic quantum optimization approach. The question of quantum speedups is not easy to answer at this point in time. In particular, the size of the devices currently available may not be large enough to fully break beyond the capabilities of classical computers. Furthermore, it is very difficult to prove that a quantum device is better than all possible classical devices at solving a particular class of problems. We are then left with comparing the quantum device against a set of classical algorithms on a particular ensemble of problems, and it is not clear if this performance will carry to more general problems, or if another more efficient classical algorithm will be developed. In this project we compared the performance of DW2 against a heuristic classical solver that has performed well in several algorithmic competitions. Even though DW2 performed well against it, it is not clear what conclusions we can draw at this point, since the classical solver was designed for general problems and not optimized to the architecture of DW2. The question of speedup has been (and still is) extensively studied by many groups around the world, and the results are still inconclusive [20]. On the other hand, the results are also promising in the sense that the performance of DW2 is on par with very optimized algorithms running on very fast hardware [20].

In this project we developed some of the tools that would be necessary for any practical application involving DW2. One of the main obstacles in using the processor is the limited connectivity of the underlying architecture that severely limits the type of problems that can be embedded exactly. It is thus unavoidable that users will have to rely on approximate or heuristic embedding techniques to solve a broader set of problems. This has its own set of drawbacks as the solutions we found are only approximate, and it is not clear a priori how good those approximations are. The sequential embedding technique developed in this project shows one possible way forward and the results are encouraging for optimization problems, but not so much for decision problems. The issue is that decision problems are implemented in DW2 by first encoding them as optimization problems: the answer to the decision problem is YES if the

minimum of a certain discrete function is equal to a certain value (usually zero), and NO otherwise. The problem with this type of formulation is that the DW2 processor does not provide a guarantee that the ground state has been found. So, if the best configuration provided by the quantum processor gives us the known minimum of the function, we know the answer to the decision problems is YES, but if it is not equal to that minimum, we cannot say that the answer is NO, because there is no guarantee that the answer found is the best possible one. We faced this issue when attempting to use DW2 to check the validity of abstract counterexamples (i.e., checking is there was a corresponding counterexample in the original system). This required solving a SAT decision problem, and the fact that we could not guarantee that the best solution was found would lead us to a wrong assertion about our problem (i.e., that a real counter example existed when that was not true).

Another important feature that was shown in parallel with the work performed for QCHECK was the quantum mechanical behavior of the D-Wave processor. We developed two approaches, one that gave evidence of a quantum signature (but fell short of being a proof), and another one that provided experimental evidence of the existence of entangled state during the annealing process. This last one constitutes a definite proof of the quantum mechanical nature of the device and it is a very important step forward in the field of practical implementations of programmable quantum devices. There is, however, a lot of work to be done to determine if these quantum effects are being exploited in a way that provides a computational speedup.

In summary, integrating adiabatic quantum optimization techniques with a model checking approach is feasible and relatively straightforward. The framework developed in this project could be easily followed in the future provided the model checking packages used provide the user with the required information. Developing newer techniques to go beyond the connectivity limitations of the DW2 device will certainly help improve the performance of the approach. And hopefully, newer generations of the device will provide the desired computational speedup and transform this proof of concept into a practical tool.

6 REFERENCES

[1] M. Johnson, et al.; “Quantum annealing with manufactured spins”; Letter, Nature, Vol. **473**, 12 May, 2011(doi:10.1038/nature 10012).

[2] C. Baier and J-P. Katoen, *Principles of Model Checking*, The MIT Press, Cambridge, Massachusetts, 2008.

[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith; “Counterexample-guided abstraction refinement for symbolic model checking”; *J. ACM*, **50**, 5 (September 2003), 752-794.

[4] F. Barahona; “On the computational complexity of Ising spin glass models”; *J. Phys. A: Math. Gen.*, **15**, 3241, 1982.

[5] A. Lucas; “Ising formulations of many NP problems”; arXiv:1302.5843.

[6] Farhi, E., et al.; “Quantum Computation by Adiabatic Evolution". arXiv:quant-ph/0001106v1.

[7] R. Harris et al.; “Experimental investigation of an eight-qubit unit cell in a superconducting optimization processor”; Phys. Rev. B, **82**, 024511 (2010).

[8] V. Choi; “Minor-embedding in adiabatic quantum computation: II. Minor-universal graph design”; arXiv:1001.3116.

[9] R. Harris, et al.; “Experimental Demonstration of a Robust and Scalable Flux Qubit”; Phys. Rev. B, **81**, 134510 (2010).

[10] R. E. Bryant; "Graph-Based Algorithms for Boolean Function Manipulation"; IEEE Transactions on Computers, C-35(8):677–691, 1986.

[11] E. Clarke, O. Grumberg, D. Long; "Model checking and abstraction"; ACM Transactions on Programming Languages and Systems, **16** (5): 1512–1542, 1992.

[12] A. Kuegel; “Improved Exact Solver for the Weighted MAX-SAT Problem”; in POS-10, EPiC Series, Vol. **8**, edited by D. L. Berre (EasyChair, 2012) pp. 15–27.

[13] Walksat Home Page: Stochastic Local Search for Satisfiability, www.cs.rochester.edu/u/kautz/walksat/. Accessed May 6th, 2014.

[14] H. Neven, V. S. Denchev, G. Rose, W. G. Macready; “QBoost: Large scale classifier training with adiabatic quantum optimization”; Asian Conference on Machine Learning (ACML), 2012.

[15] S. Bieniawski, I. Kroo, D. Wolpert; “Discrete, continuous, and constrained optimization using collectives”; 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, 30 August - 1 September, 2004, Albany, New York.

[16] T. M. Cover, J. A. Thomas; *Elements of Information Theory, 2nd Edition*; Wiley Series in Telecommunications and Signal Processing, Wiley-Interscience, Hoboken, New Jersey, 2006.

[17] S. Miller, E. Anderson, L. Wagner, M. Whalen, M. Heimdahl; “Formal verification of flight critical software”; AIAA Guidance, Navigation and Control Conference and Exhibit, San Francisco, August 15-18, 2005.

[18] S. Boixo, T. Albash, F.M. Spedalieri, N. Chancellor, D. A. Lidar; “Experimental signature of programmable quantum annealing”; Nature Communications **4**, Article number: 2067, published online 28 June 2013.

[19] T. Lanting, A.J. Przybysz, A. Yu. Smirnov, F.M. Spedalieri, M.H. Amin, A.J. Berkley, R. Harris, F. Altomare, S. Boixo, P. Bunyk, N. Dickson, C. Enderud, J.P. Hilton, E. Hoskinson, M.W. Johnson, E. Ladizinsky, N. Ladizinsky, R. Neufeld, T. Oh, I. Perminov, C. Rich, M.C. Thom, E. Tolkacheva, S. Uchaikin, A.B. Wilson, G. Rose; “Entanglement in a quantum annealing processor”; arXiv:1401.3500, accepted for publication in Physical Review X, 2014.

[20] T. Rønnow, Z. Wang, J. Job, S. Boixo, S. Isakov, D. Wecker, J. Martinis, D. Lidar, M. Troyer; “Defining and detecting quantum speedup”; arXiv:1401.2910, 2014.

7 APPENDIX A – Publications and Presentations

The following two publications, regarding the quantum nature of the D-Wave device, were written in parallel with the work performed for the QCHECK project. Although not a part of the proposed research, these results are included to complement and give better context to the results presented in this report.

- Quantum Signature

  S. Boixo, T. Albash, F.M. Spedalieri, N. Chancellor, D. A. Lidar; "Experimental signature of programmable quantum annealing"; Nature Communications **4**, Article number: 2067, published online 28 June 2013.

- Quantum entanglement

  T. Lanting, A.J. Przybysz, A. Yu. Smirnov, F.M. Spedalieri, M.H. Amin, A.J. Berkley, R. Harris, F. Altomare, S. Boixo, P. Bunyk, N. Dickson, C. Enderud, J.P. Hilton, E. Hoskinson, M.W. Johnson, E. Ladizinsky, N. Ladizinsky, R. Neufeld, T. Oh, I. Perminov, C. Rich, M.C. Thom, E. Tolkacheva, S. Uchaikin, A.B. Wilson, G. Rose; "Entanglement in a quantum annealing processor"; arXiv:1401.3500, accepted for publication in Physical Review X, 2014.

8 APPENDIX B – Description of CEGAR-DW2 integration code

In this appendix we provide a brief description of the main code components needed to integrate the CEGAR approach with the DW2 capabilities. As discussed in Section 4.4, we were forced to construct a simple model checking program (based on freely available BDD libraries) because publicly available model checking packages did not provide the user with low level information needed to construct the ILPs required during the CEGAR process. This code (and supporting files) can be accessed at http://www.isi.edu/people//fspedali/QCHECK_code

To run the integrated CEGAR process, we call a Perl script named `cegar-solver-loop.pl`. This script takes as an argument the name of the file where the system is described in AIG format. The CEGAR process is then applied: an abstraction is generated, the model checker is called to check the required property, and if a counterexample is found, it is checked using a regular SAT solver. When the counter example is found to be spurious, an ILP is generated to solve the Minimal Separating Set problem that would allow us to refine the abstraction. Once the ILP is generated, it is written to a temporary file. Then the DW2 solver is called (see the line

```
my $answer =`matlab -nodisplay -nosplash -r
"SolverILP('$tmpfile')"  | tail -n +11`;
```

in `cegar-solver-loop.pl`). This command calls Matlab and runs the function `SolverILP`, that takes the name of the file where the ILP is stored as an argument, and then calls other functions that write the ILP as a QUBO problem and solves it using the heuristic embedding coded in the function `SEBREMforQUBO`. The answer is sent to the standard output, where the Perl script grabs it and continues with the CEGAR process, refining the abstraction according with the solution of the ILP. The code can be run in verbose mode, where the successive ILPs solved are shown. A detailed transcript of one such run was presented in Section 4.5.

9 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

σ	Pauli operator
2-SAT	2-Satisfiability problem
AIG	And Inverter Graph
AIGER	Format for And-Inverter Graphs
API	Application Programming Interface
AX-MAXSAT	Exact MAX-SAT solver
BDD	Binary Decision Diagram
BMC	Bounded Model Checking
CE	Counter Example
CEGAR	Counter-example Guided Abstraction Refinement
CMU	Carnegie Mellon University
CNF	Conjunctive Normal Form
CTL	Computational Tree Logic
CUDD	CU Decision Diagram package
DIMACS	Computer-readable format for Boolean satisfiability problems
DW1	D-Wave One
DW2	D-Wave Two
FBK	Fondazione Bruno Kessler
FCS	Flight Control System
h	Vector of local magnetic fields
ILP	Integer Linear Program
ISI	Information Sciences Institute
J	Matrix of inter qubit couplings
`laig`	Lisp-flavored AIG
LTL	Linear Temporal Logic
MAX-2-SAT	Maximum 2-Satisfiability problem
MaxWalkSAT	WalkSAT variant for weighted SAT problem solver
MC	Model Checking

NP	Non-deterministic Polynomial class of decision problems
NP-hard	Computational complexity class that contains the class NP
NuSMV	New Symbolic Model Checker for System Verification
NuXMV	New Symbolic Model Checker for the Analysis of Synchronous finite-state and infinite-state Systems
OBDD	Ordered Reduced Binary Decision Diagram package
QCHECK	A Quantum Computing Approach to Model Checking for Advanced Manufacturing Problems
QUBO	Quadratic Unconstrained Binary Optimization
rf-SQUID	Radio Frequency Superconducting Quantum Interference Device
SAT	Boolean satisfiability
SEBREM	Sequential Embedding By Relative Entropy Minimization
SMV	Symbolic Model Checker for System Verification
SQUID	Superconducting Quantum Interference Device
WalkSAT	Boolean satisfiability problem solver
XML	Extended Markup Language