Forum TERATEC 23

Unlock the future

31 MAI & 1er JUIN 2023 • Au Parc Floral, Paris

Un événement organisé par infoprodigital Teratec

DE LA RECHERCHE À L'INDUSTRIE

Quantum Software Stack, a Software Science perspective

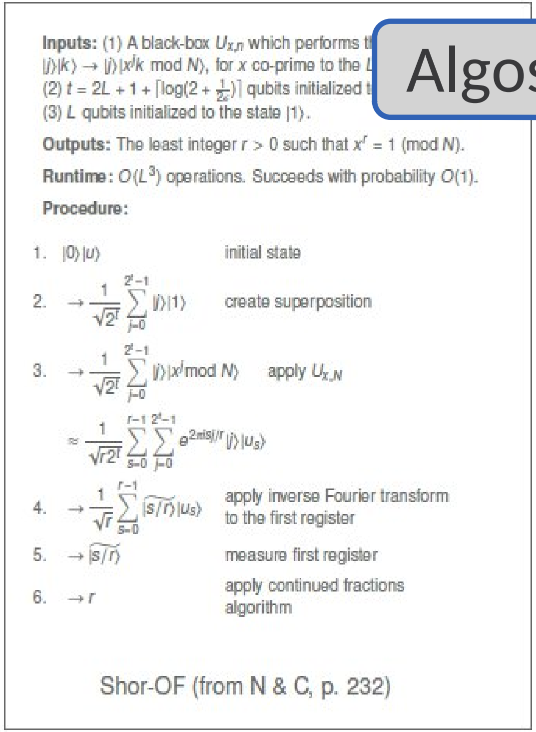
Christophe Chareton, Sébastien Bardin

Goals of this talk

Provide an overview of quantum programming languages and the related software stack

Understand what is at stake and the underlying scientific challenges

Quantum computing: our challenge



Inputs: (1) A black-box $U_{x,N}$ which performs th
$|j\rangle|k\rangle \rightarrow |j\rangle|x^j k \bmod N\rangle$, for x co-prime to the L
(2) $t = 2L + 1 + \lceil \log(2 + \frac{1}{2\epsilon}) \rceil$ qubits initialized t
(3) L qubits initialized to the state $|1\rangle$.

Outputs: The least integer $r > 0$ such that $x^r = 1 \pmod N$.

Runtime: $O(L^3)$ operations. Succeeds with probability $O(1)$.

Procedure:

1. $|0\rangle|u\rangle$ — initial state
2. $\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|1\rangle$ — create superposition
3. $\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|x^j \bmod N\rangle$ — apply $U_{x,N}$
   $\approx \frac{1}{\sqrt{r2^t}} \sum_{s=0}^{r-1} \sum_{j=0}^{2^t-1} e^{2\pi i sj/r}|j\rangle|u_s\rangle$
4. $\rightarrow \frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |\widetilde{s/r}\rangle|u_s\rangle$ — apply inverse Fourier transform to the first register
5. $\rightarrow \widetilde{s/r}$ — measure first register
6. $\rightarrow r$ — apply continued fractions algorithm

Shor-OF (from N & C, p. 232)

Algos

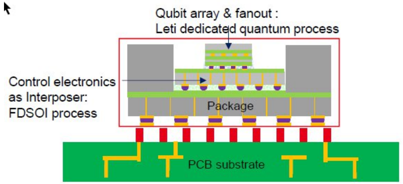
How ?

Hardware

Quantum Stack, software and architectural

Practical algorithms

QUANTUM STACK



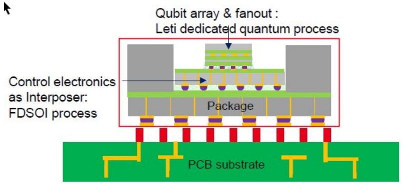
Hardware interface & control

Quantum Stack, software and architectural

Practical algorithms

QUANTUM STACK

- effective programming
- correct & efficient programs
- portable, maintainable



Hardware interface & control

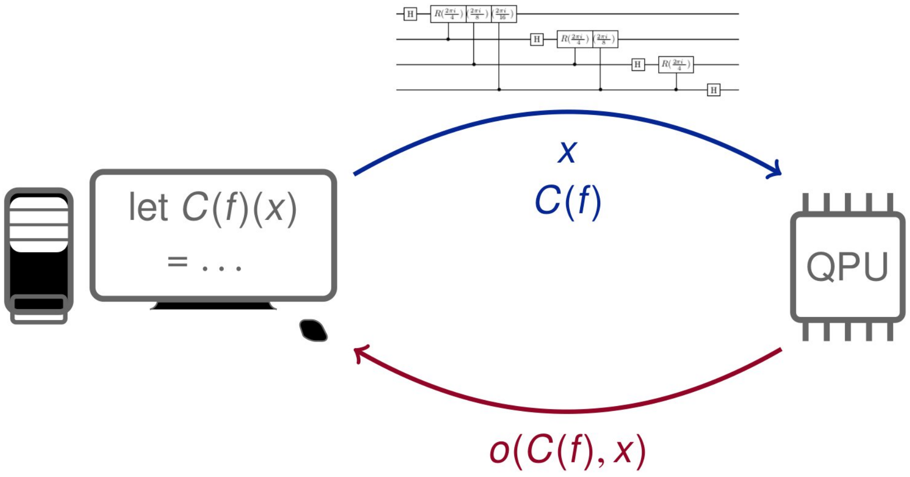
The hybrid model

A quantum co-processor (QPU), controlled by a classical computer

- classical control flow
- CPU ⇒ QPU : quantum computing requests, sent to the QPU

→structured sequenced of instructions: **quantum circuits**

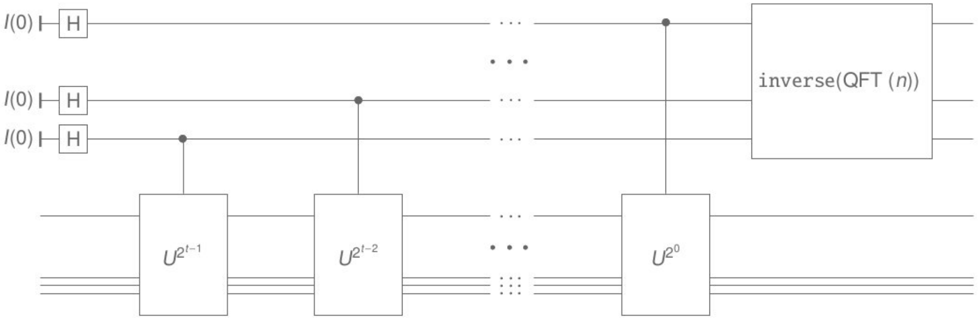
- QPU ⇒ CPU: **probabilistic** computation results (**classical** information)



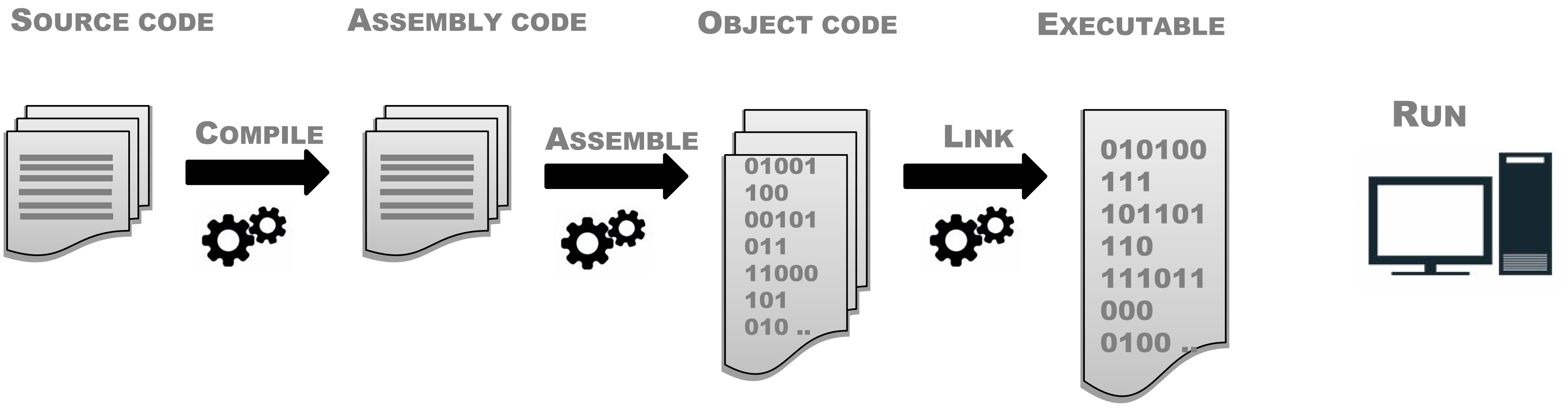
The hybrid model

- circuit
- circuit generator : input ⇒ circuit

```
def qft_rotations(circuit, n):
    """Performs qft on the first n qubits in circuit (without swaps)"""
    if n == 0:
        return circuit
    n -= 1
    circuit.h(n)
    for qubit in range(n):
        circuit.cp(pi/2**(n-qubit), qubit, n)
    # At the end of our function, we call the same function again on
    # the next qubits (we reduced n by one earlier in the function)
    qft_rotations(circuit, n)
```

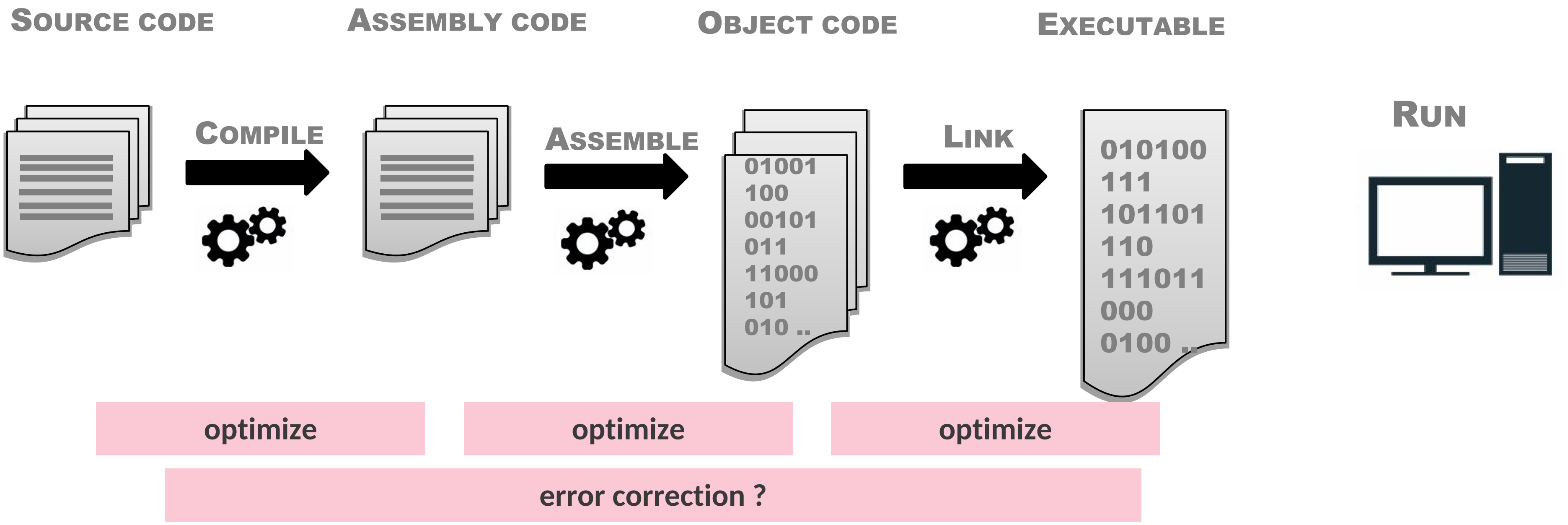


Reminder : the classical software stack (101)



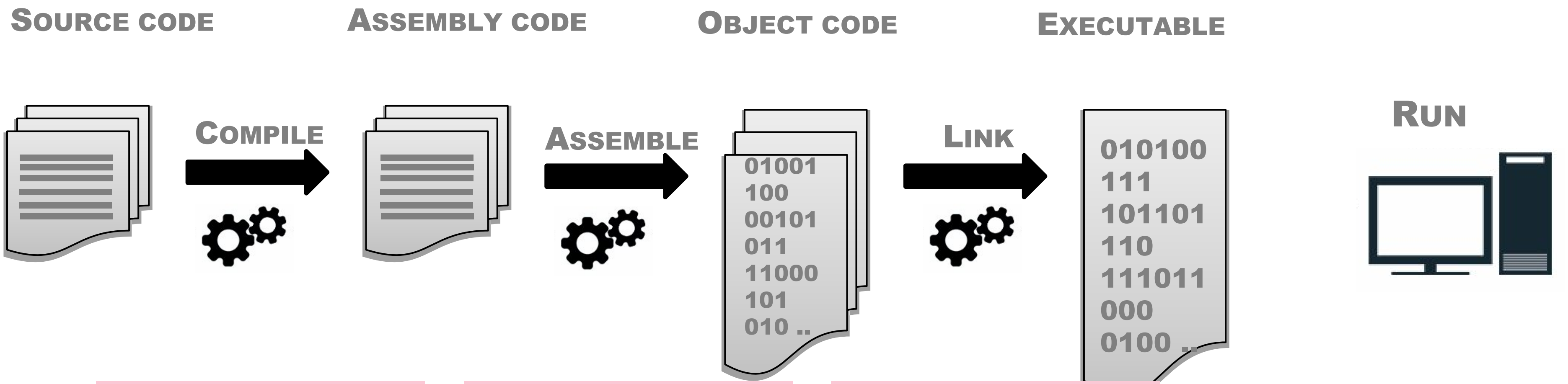
Basic compiler chain is not enough

Reminder : the classical software stack (102)



Reminder : the classical software stack (state of the art)

Certified compilation



optimize

optimize

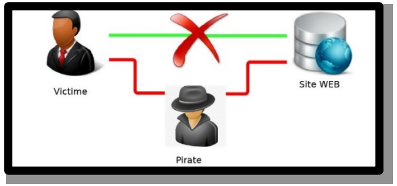
optimize

error correction ?

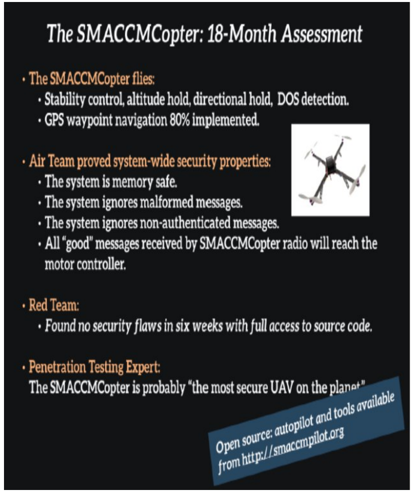
Type checking, sanity checks

Testing, program checking

Key ingredient :
math inside (logic, semantic)



TLS 1.3



The SMACCMCopter: 18-Month Assessment

- The SMACCMCopter flies:
  - Stability control, altitude hold, directional hold, DOS detection.
  - GPS waypoint navigation 80% implemented.
- Air Team proved system-wide security properties:
  - The system is memory safe.
  - The system ignores malformed messages.
  - The system ignores non-authenticated messages.
  - All "good" messages received by SMACCMCopter radio will reach the motor controller.
- Red Team:
  - Found no security flaws in six weeks with full access to source code.
- Penetration Testing Expert:
  The SMACCMCopter is probably "the most secure UAV on the planet"

Open source: autopilot and tools available from http://smaccmpilot.org

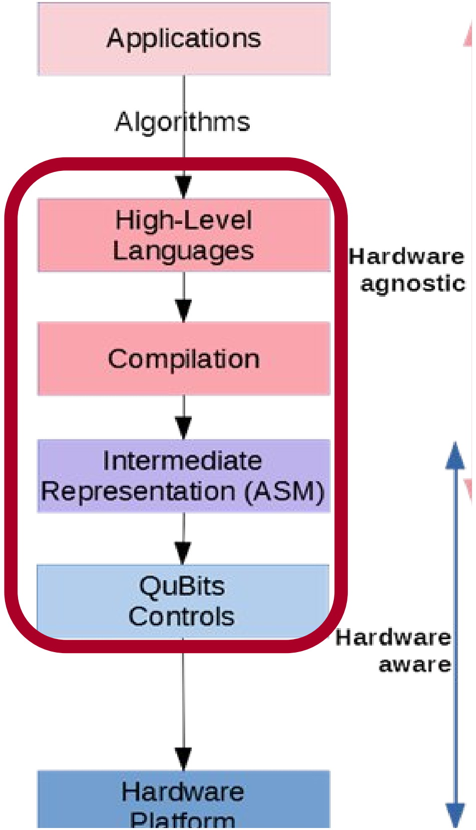
Success in the classical world!

Key ingredient :
math inside (logic, semantic)

Challenge : Quantum Software Stack

Key ingredient :
math inside (logic, semantic)

- **How to productively write quantum programs?**
- **How to ensure their quality?**
- **How to compile them efficiently?**
- **How much hardware-agnostic can we be?**
- **How to ensure correctness all along?**



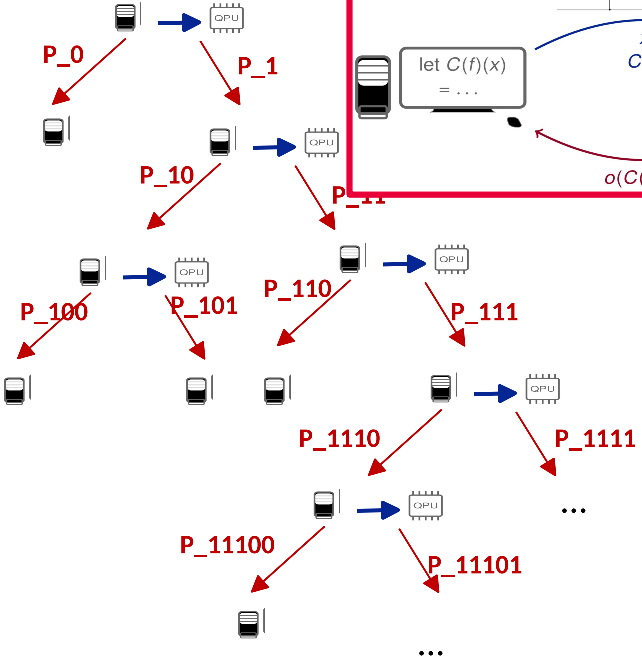
Research in Software Science has some ~~answers~~ insights there

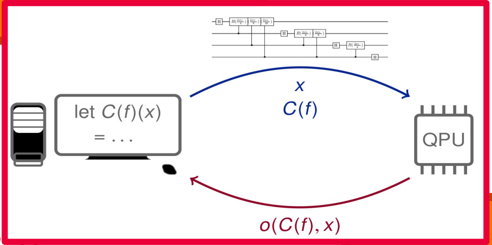
Outline

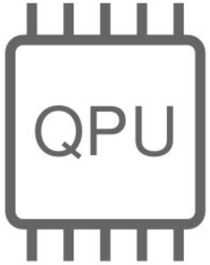
- Context
- Quantum programming is tricky
- Focus: languages
- Focus: testing & validation
- Conclusion

Quantum computing is tricky

- **Probabilistic** executions





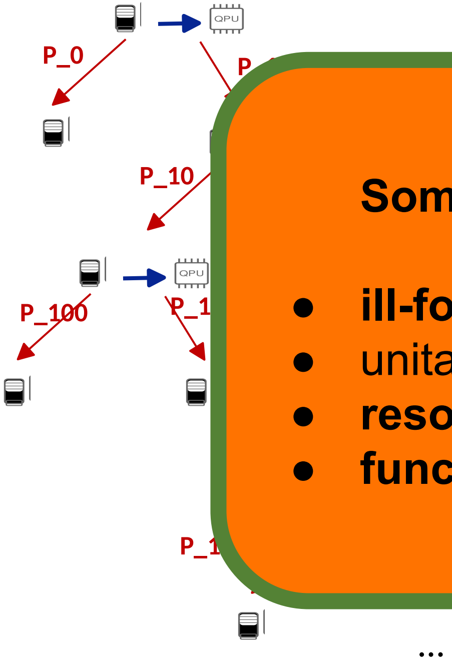


Within nodes : Some *strange* rules:

- **unitarity/no cloning**
- **destructive measure**
- restricting set of operations ("unitary")

Quantum computing is tricky

- **Probabilistic** executions



Some traps to avoid… Eg.:

- **ill-formed** dynamic circuit building
- unitarity → **subcircuit control**
- **resource** requirements
- **functionality**

ors

ations (“unitary”)

Bugs: The case for subcircuit control

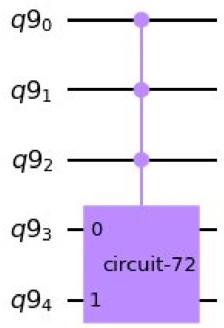
QBRICKS

- Subcircuit control : a **unavoidable** features, present in any algorithm
- Problem : a controlled gate should not modify its control qubit **(unitarity)**
- Blind spot. Eg. Qiskit: control extends the register it applies on.

requires{c= a +b}

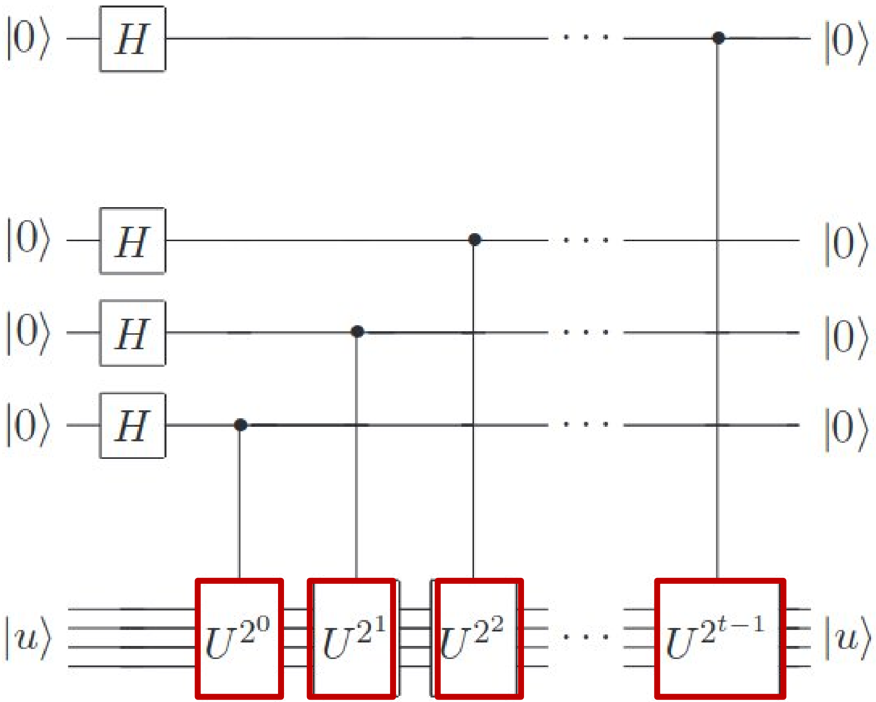
```
qc1 = QuantumCircuit(a)
custom = qc1.to_gate().control(b)

qr = QuantumRegister(c)
qc2 = QuantumCircuit(qr)
qc2.append(custom, qr)
qc2.draw()
```



Bugs : guarantee required resources

Phase estimation, N&C :



- exponential sequence composition
- -> beware of exponential complexity
- loses any potential quantum advantage!!
- Interprete U^k as

  "a gate simulating U^k"

requires{number_of_gates(U^k)=O(P(n)) }

Traps: unintuitive computation/high risk for bugs

Example of the Quantum Fourier Transform (Qiskit documentation)

```
def qft_rotations(circuit, n):
    """Performs qft on the first n qubits in circuit (without swaps)"""
    if n == 0:
        return circuit
    n -= 1
    circuit.h(n)
    for qubit in range(n):
        circuit.cp(pi/2**(n-qubit), qubit, n)
    # At the end of our function, we call the same function again on
    # the next qubits (we reduced n by one earlier in the function)
    qft_rotations(circuit, n)
```

Simulated double loop

Recursive definition

$$|j\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \omega_N^{jk} |k\rangle$$

Interpretation as a sum of vectors in a complex vectorial space

Outline

- Context
- Quantum programming is tricky
- Focus: languages
- Focus: testing & validation
- Conclusion

Q#
(Microsoft)

SILQ

Qiskit
(IBM)

ProjectQ
(IBM+ETH)

Cirq
(Google)

myQLM
(Atos)

QLC

LIQUi|>
(Microsoft)

Qwire

QBRICKS

Scaffold

Quipper

ProtoQuipper

Sqir

qPCF

So, problem solved ?

Industrial languages: go fast and take the lead

- Hybrid model
- In support of future machines
- Often imperative programming (Python: Circ, Qiskit, aQASM, ProjectQ)
  + some functional (F#: Q#,LiquiD)

ProjectQ

(IBM+ETH)

aQASM

(Atos)

Cirq

(Google)

- iterative ad hoc design rather than minimal principled design
- Very few guarantees on the produced code
- how to ensure good performance?

LIQUi|>

(Microsoft)

- **Good support, quick evolution**
- **Development of users communities**
- **Industrial means: large libraries**

Q#

(Microsoft)

ProjectQ
(IBM+ETH)

aQASM
(Atos)

Cirq
(Google)

Leverage Programming Language Research to the Quantum Case

LIQUi|>
(Microsoft)

Q#
(Microsoft)

Scaffold

Qwire

QLC

SOME RECENT ACHIEVEMENTS

- languages with **formal verif** (Qbricks/CEA-LMF, Sqir/ Univ. of Maryland, QHL/Tsinghua Univ.)
- no-cloning: by **design** (Qbricks) or **linear types** (Qwire/Univ. of Pennsylvania, Sqir)
- Well-formedness : **dependent types** (ProtoQuipper/Dalhousie Univ.), **contracts** (Qbricks)
- Automated **uncomputation** (SILQ/ETH Zurich)...

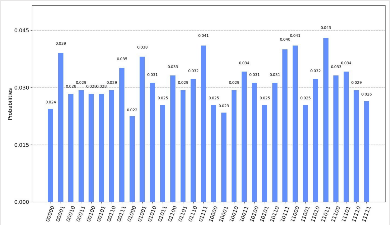
CF

Quipper

ProtoQuipper

- Context
- Quantum programming is tricky
- Focus: languages
- Focus: testing & validation
- Conclusion

Standard verification methods fail

<table>
<tr><td>Assertion checking?</td><td>Requires (destructive) measurement</td></tr>
<tr><td>Tests?</td><td>Requires runs in exponential number

</td></tr>
<tr><td>Simulation?</td><td>As far as we don’t need a Quantum Computer</td></tr>
</table>

Formal verification vs. Standard debug

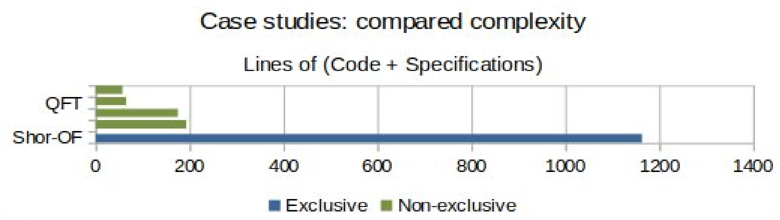
Testing/Assertion checking	Formal verification
tested instance	**any** instance
based on executions/simulations	**static** analysis, **no need to execute**
bounded parameters	**scale insensitive**
non deterministic programs : **statistical arguments**	absolute, **mathematical guarantee**

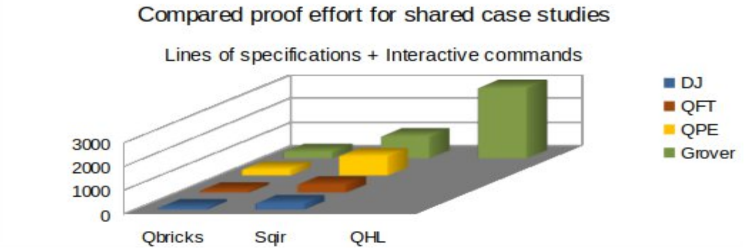
Build on **best practice** of formal verification for the classical case and **tailor them to the quantum** case

Focus : QBRICKS

MAJOR ACHIEVEMENTS

- a core development framework for **parametrized verified quantum programming**
- **first ever verified implementation of Shor order finding algorithm** (95% proof automation),





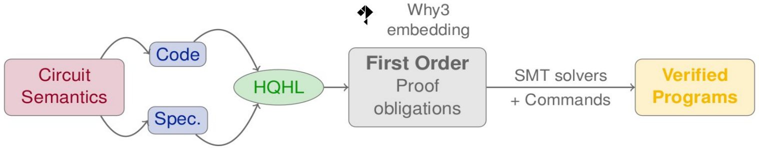
An Automated Deductive Verification Framework for Circuit-building Quantum Programs

Christophe Chareton[1,2,✉], Sébastien Bardin[2], François Bobot[2], Valentin Perrelle[2], and Benoît Valiron[1]

[1] LMF, CentraleSupélec, Université Paris-Saclay, Gif-sur-Yvette, France
firstname.lastname@lri.fr
[2] CEA, LIST, Université Paris-Saclay, Palaiseau, France
firstname.lastname@cea.fr

Abstract. While recent progress in quantum hardware open the door for significant speedup in certain key areas, quantum algorithms are still hard to implement right, and the validation of such quantum programs is a challenge. In this paper we propose QBRICKS, a formal verification environment for circuit-building quantum programs, featuring both parametric specifications *and* a high degree of proof automation. We propose a logical framework based on first-order logic, and develop the main tool we rely upon for achieving the automation of proofs of quantum specification: PPS, a parametric extension of the recently developed path sum semantics. To back-up our claims, we implement and verify parametric versions of several famous and non-trivial quantum algorithms, including the quantum parts of *Shor's integer factoring*, quantum phase estimation (QPE) and Grover's search.

Keywords: deductive verification, quantum programming, quantum circuits



Focus : QBRICKS

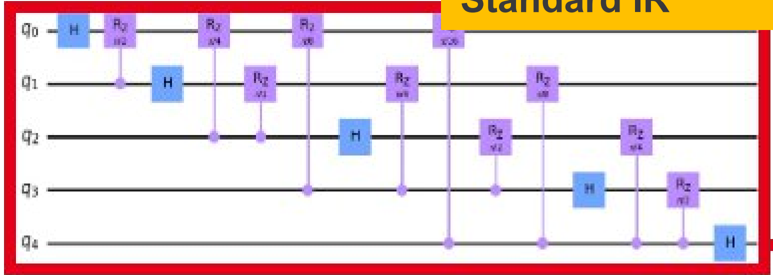
C_1 → C_2 → .... → C_k → C_{Oqasm}

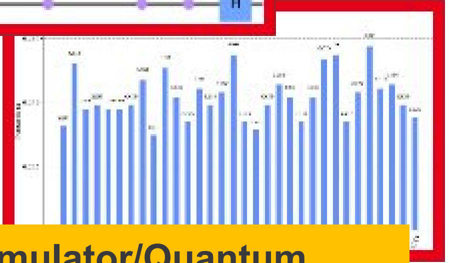
Circuit combiner deletion
- parralelism
- quantum control

Gate transformation

```
|| qft || (qreg qr)
      circ qr ->
      for q in range(len(qr)) {
         H(qr[q])
         for i in range(qr[q+1..-1]) {
            with control qr[i+1] (RZ(i-q, qr[q]))
      return
```

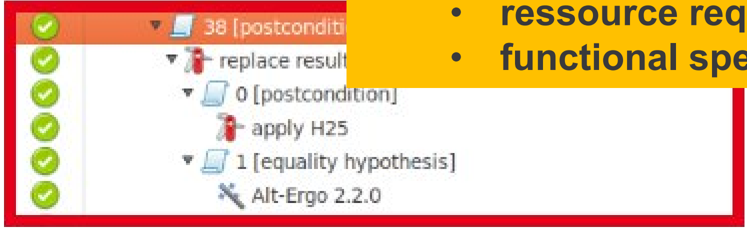
Standard IR





Simulator/Quantum machine

- **Proofs**
  - **well-formedness**
  - **ressource requirements**
  - **functional specs**



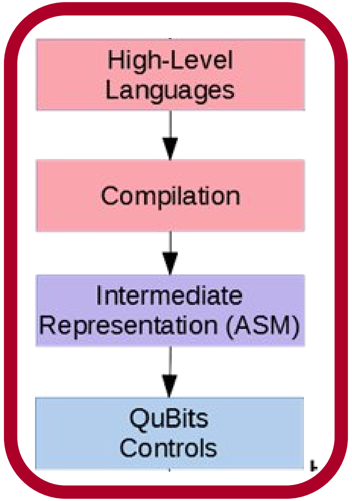
Outline

- **Context**
- **Quantum programming is tricky**
- **Focus: languages**
- **Focus: testing & validation**
- **Conclusion**

Quantum software science does matter

- Build the bridge between Quantum Algo. and Quantum Hardware
- Several challenges ahead

- effective programming
- correct & efficient programs
- portable, maintainable



- Software science principles can help
  - Leverages lessons from classical case
  - Still, push the methods to their edge
  - // could have impact on classical software in turn
- Research in progress

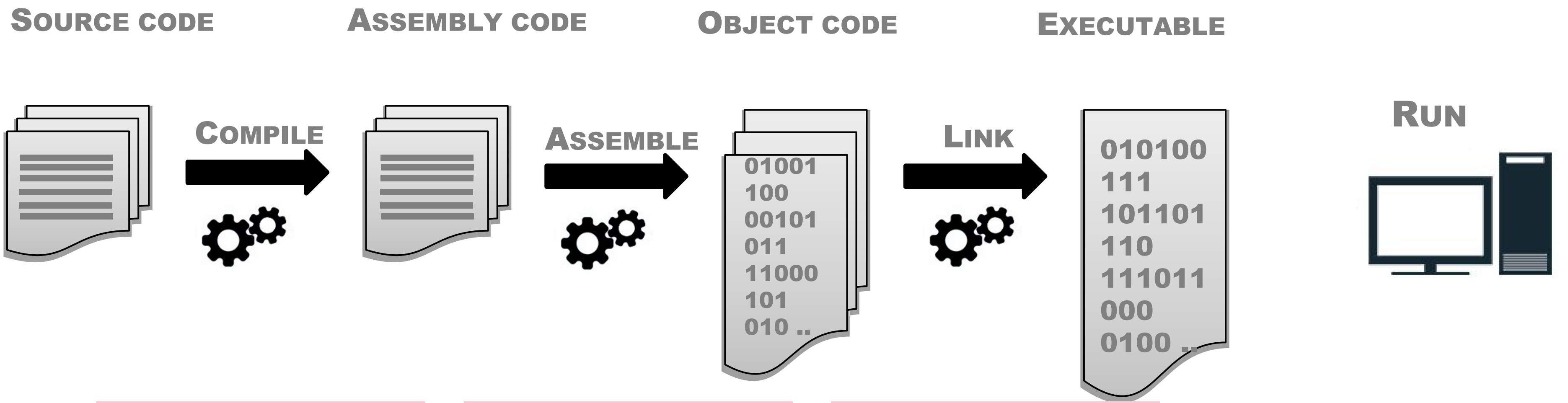
Langage semantic & design

Language-based vs side analyzers

Tradeoff automation - expressiveness

Genericity vs. specialization

Back to basic



optimize

optimize

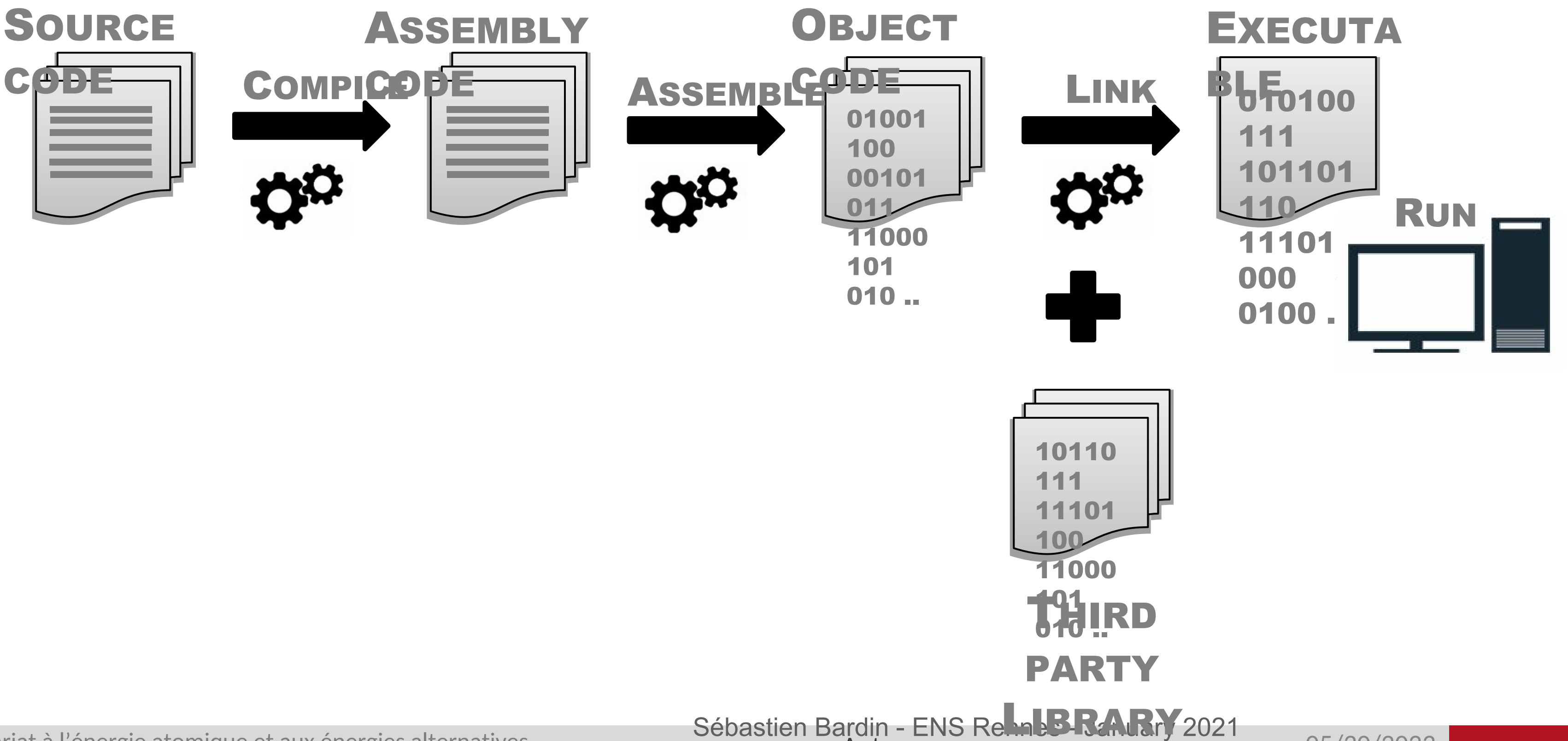
optimize

Type checking, sanity checks

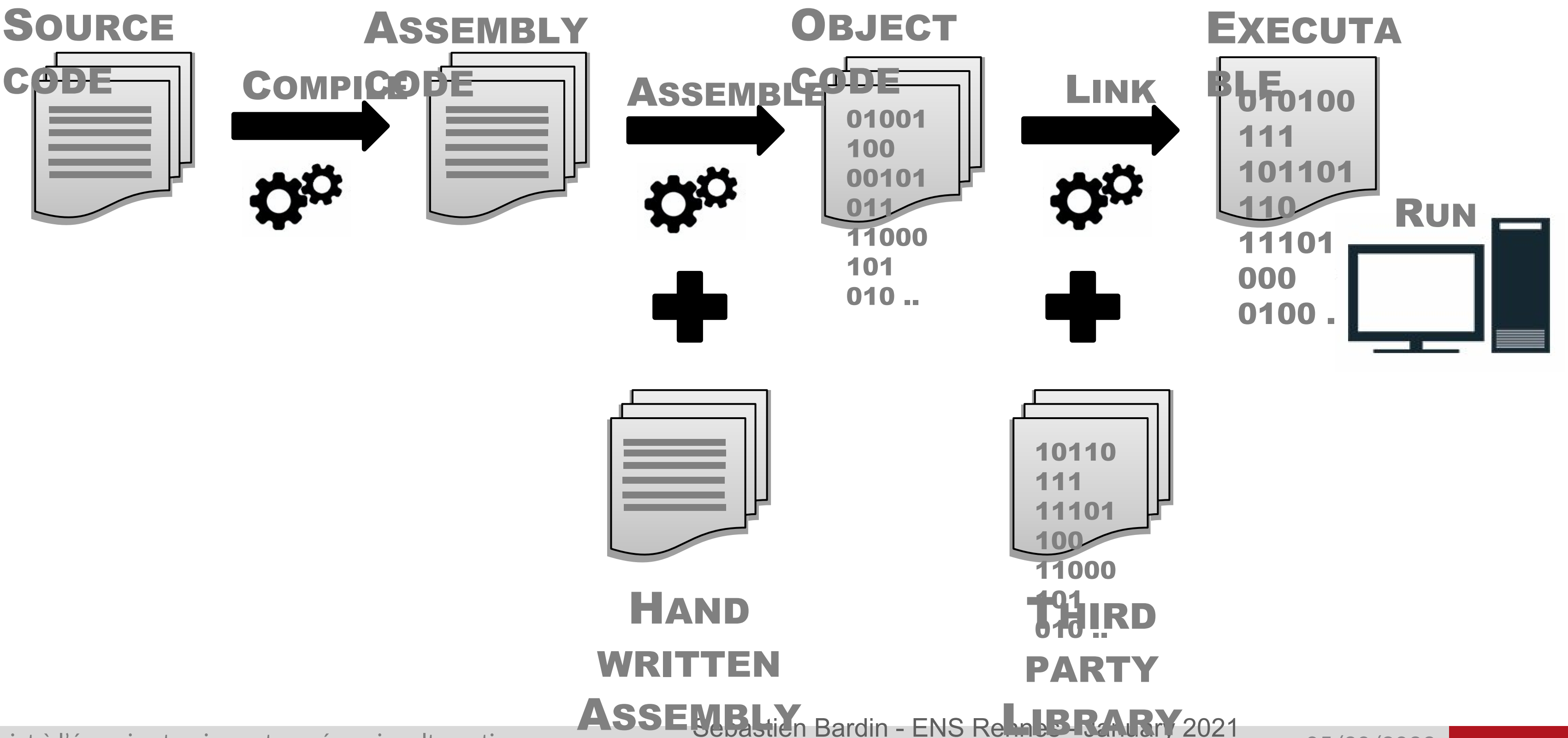
Testing, program checking

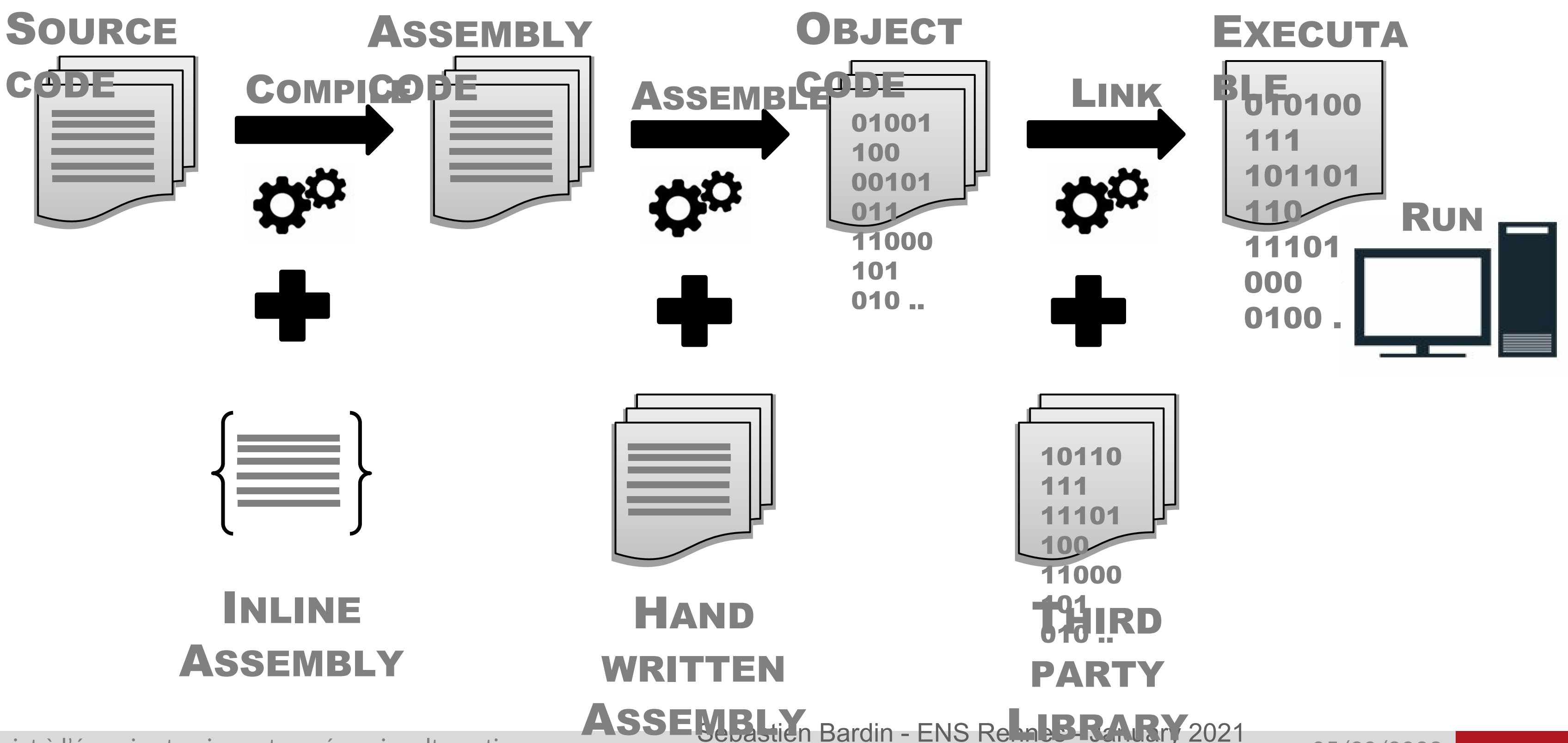
Certified compilation

Back to basic



Back to basic




Source code
Compile
Assembly code
Assemble
Object code
01001 100 00101 011 11000 101 010 ..
Link
Executable
010100 111 101101 110 11101 000 0100 .
Run
Inline Assembly
Hand written Assembly
10110 111 11101 100 11000 101 010 ..
Third party library

Quantum programming and verification : take away

- Quantum computing is **tricky**
- Standard debugging methods **fail**
- We promote and develop **formal reasoning** and static analysis

Qbricks development

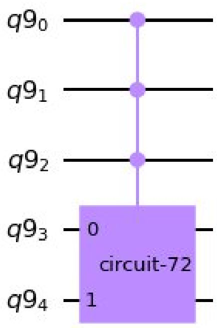
- **POC** for formal verification
- **Specs/prototypes and road map** for further developments

Bugs: The case for subcircuit control

- Subcircuit control : a **unavoidable** features, present in any algorithm
- Problem : a controlled gate should not modify its control qubit **(unitarity)**
- Blind spot. Eg. Qiskit: control extends the register it applies on.

```
qc1 = QuantumCircuit(a)
custom = qc1.to_gate().control(b)

qr = QuantumRegister(c)
qc2 = QuantumCircuit(qr)
qc2.append(custom, qr)
qc2.draw()
```

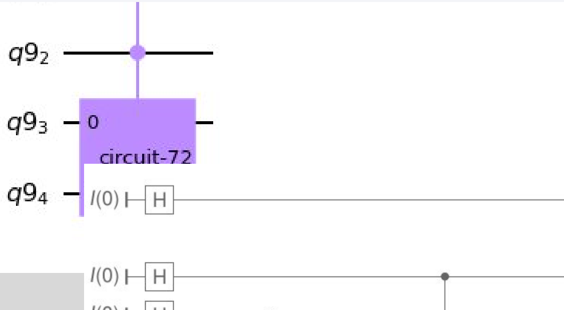


Bugs: The case for subcircuit control

- Subcircuit control : a **unavoidable** features, present in any algorithm
- Problem : a controlled gate should not modify its control qubit **(unitarity)**
- Blind spot. Eg. Qiskit: control extends the register it applies on.

requires{c= a +b}

```
qc1 = QuantumCircuit(a)
def qft_rotations(circuit, n):
    """Performs qft on the first n qubits in circ
    if n == 0:
        return circuit
    n -= 1
    circuit.h(n)
    for qubit in range(n):
        circuit.cp(pi/2**(n-qubit), qubit, n)
    # At the end of our function, we call the sam
    # the next qubits (we reduced n by one earlie
    qft_rotations(circuit, n)
```

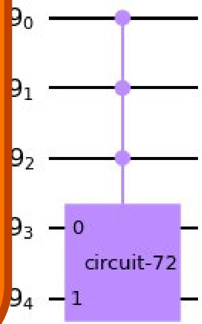


Bugs: The case for subcircuit control

- Subcircuit control : a **unavoidable** features, present in any algorithm
- Problem : a controlled gate should not modify its control qubit **(unitarity).**
- Blind spot. Eg. Qiskit: control extends the register it applies on.

```
qc1 = QuantumCircuit(a)
custom = qc1.to_gate().control(b)

qr = QuantumRegister(c)
qc2 = QuantumCircuit(qr)
qc2.append(custom, qr)
qc2.draw()
```



We aim at offering no cloning guarantees +

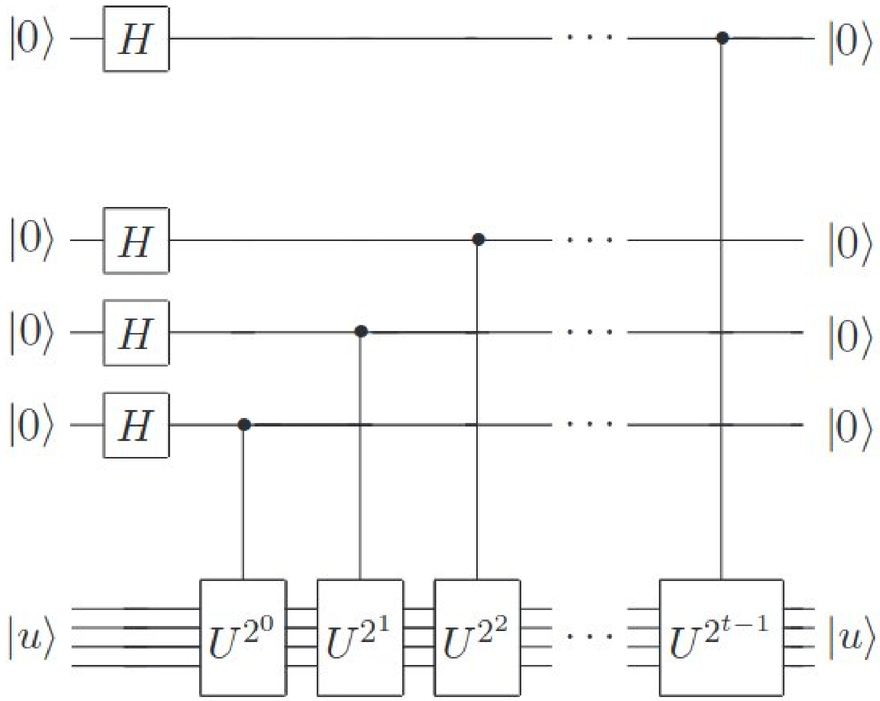
- **language** : intuitively

  eg. : with control c apply U

- **formalization** : convenient representation
- **compilation**: generic interpretation

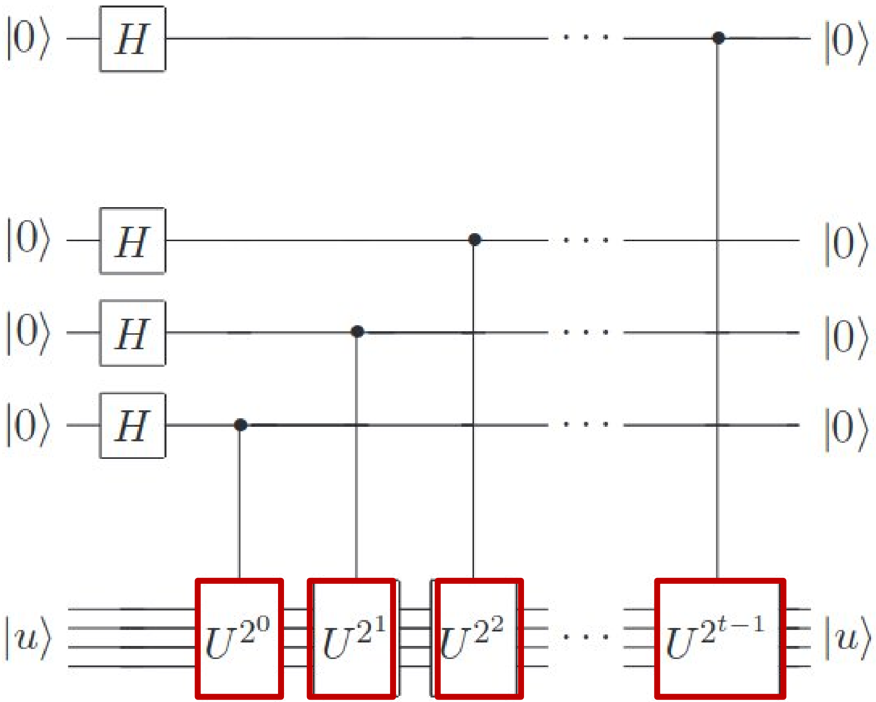
Bugs : guarantee required resources

Phase estimation, N&C :



Bugs : guarantee required resources

Phase estimation, N&C :

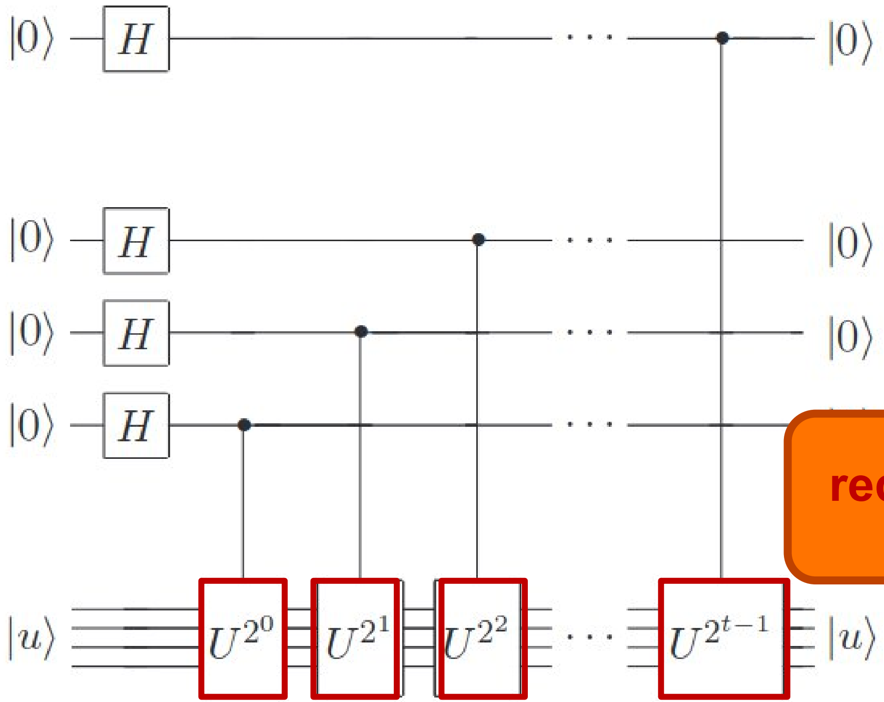


- **exponential** sequence composition
- **loses any potential quantum advantage!!**
- Interprete U^k as

  "a gate simulating U^k*"*

Bugs : guarantee required resources

Phase estimation, N&C :

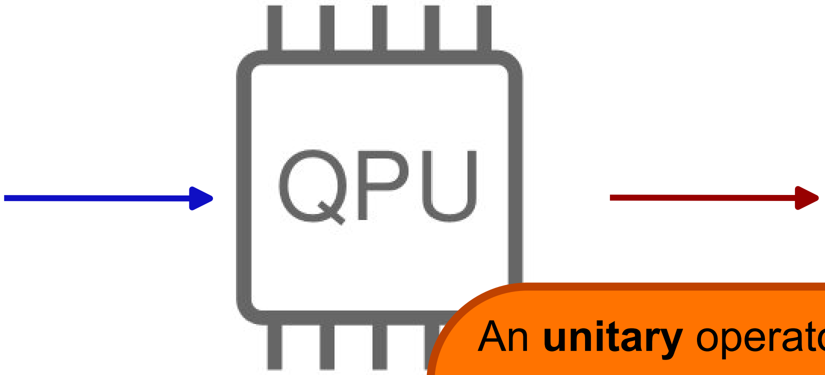


- exponential sequence composition
- -> exponential complexity
- loses any potential quantum advantage!!
- Interprete U^k as

  "a gate simulating U^k"

requires{number_of_gates(U^k)=O(P(n)) }

Bugs : functional correctness



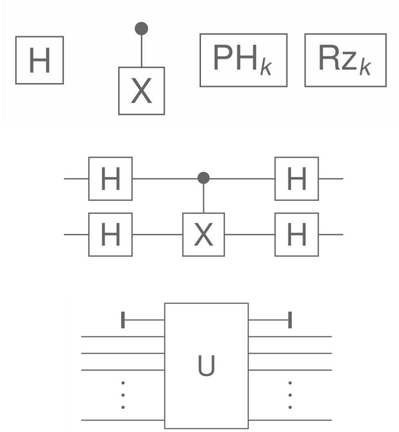
An **unitary** operator over a **complex vectorial space** of **exponential** dimension:

→ Is it (really) a **correct encoding** of our “real life” problem?

- **Prime factor** decomposition (Shor)
- Finding an antecedent through an **integer function** (Grover)
- **Optimization** …

Programming framework: QBRICKS-DSL

- **Functional** programming + Why3 embedding
- Unitary programs
- Circuit as objects → no cloning **by construction**
- **Deductive verification** → use of **contracts** (well formedness + functional correctness + complexity)



- A minimal set of primitive functions
  - elementary gates
  - compositions : parallel/sequence
  - ancilla creation/anihilation
- Circuits as elementary algebraic objects

→ anything concerning **qbits** and state evolution is delegated to the **specifications**

- derived high-level combinators: inversion, control, qbit permutations, etc

QBRICKS / Demonstration

Algorithm: Quantum phase estimation

Inputs: (1) A black box wich performs a controlled-U^j operation, for integer j, (2) an eigenstate $|u\rangle$ of U with eigenvalue $e^{2\pi i \varphi_u}$, and (3) $t = n + \lceil \log\left(2 + \frac{1}{2\epsilon}\right) \rceil$ qubits initialized to $|0\rangle$.

Outputs: An n-bit approximation $\widetilde{\varphi_u}$ to φ_u.

Runtime: $O(t^2)$ operations and one call to controlled-U^j black box. Succeeds with probability at least $1 - \epsilon$.

Procedure:

1.	$|0\rangle|u\rangle$	initial state
2.	$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|u\rangle$	create superposition
3.	$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle U^j|u\rangle$	apply black box
	$= \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} e^{2\pi i j \varphi_u}|j\rangle|u\rangle$	result of black box
4.	$\rightarrow |\widetilde{\varphi_u}\rangle|u\rangle$	apply inverse Fourier transform
5.	$\rightarrow \widetilde{\varphi_u}$	measure first register

Specification preamble

- input parameters + preconditions
- post-conditions:
  - functional
  - complexity

Body

- sequence of quantum operations,
- intermediate system state postconditions

QBRICKS / Demonstration

Decorated code

```
let  function apply_black_box (circ:circuit)(k n:int)
                              (ghost y:matrix complex)(ghost theta:complex)

 = sequence (create_superposition k n) (black_box circ k y theta)
```

Functional programming

parameters -> quantum circuits

QBRICKS / Demonstration

Decorated code

```
let  function apply_black_box (circ:circuit)(k n:int)
                              (ghost y:matrix complex)(ghost theta:complex)
   requires{n=k+width circ}
   requires{real_ theta}
   requires{0 < k < n}
   requires{is_a_ket_l y (n-k)}
   requires{eigen circ y (real_to_ang theta)}

   ensures{width result = n}
   ensures{ancillas result =0}
   ensures{size result<=n+k*size circ}

   ensures{path_sem result  (kron (ket k 0)y) =
           (kron (pow_inv_sqrt_2 k *..   ket_sum (n_bvs k)
            (fun x -> black_box_coeff  theta x *..  (bv_to_ket x)) k) y)}
 = sequence (create_superposition k n) (black_box circ k y theta)
```

Functional programming

`parameters -> quantum circuits`

Specifications

- preconditions
- complexity specifications
- functional assertions

QBRICKS / Demonstration

Proof obligations generation, via Why3 interface

```
let  function apply_black_box (circ:circuit)(k n:int)
                              (ghost y:matrix complex)(ghost theta:complex)
   requires{n=k+width circ}
   requires{real_ theta}
   requires{0 < k < n}
   requires{is_a_ket_l y (n-k)}
   requires{eigen circ y (real_to_ang theta)}

   ensures{width result = n}
   ensures{ancillas result =0}
   ensures{size result<=n+k*size circ}

   ensures{path_sem result  (kron (ket k 0)y) =
            (kron (pow_inv_sqrt_2 k *..    ket_sum (n_bvs k)
             (fun x -> black_box_coeff  theta x *..  (bv_to_ket x)) k) y)}

 = sequence (create_superposition k n) (black_box circ k y theta)
```

- VC apply_black_box [VC for apply_black_box]
  - split_vc
    - 0 [precondition]
    - 1 [precondition]
    - 2 [precondition]

⋮ ⋮

    - 36 [precondition]
    - 37 [postcondition]
    - 38 [postcondition]
- VC phase_estimation [VC for phase_estimation]
- VC pe_measure [VC for pe_measure]
- VC best_appr [VC for best_appr]
- VC delta [VC for delta]

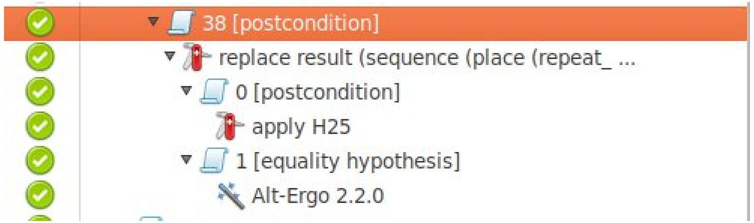
```
goal VC apply black box :
  path sem result (kronecker (ket k 0) y)
  = kronecker
    (pow_inv_sqrt_2 k
     *.. ket_sum_l (n_bvs k)
         (fun (x:bitvec) -> black_box_coeff theta x *.. bv_to_ket x) k)
    y
```

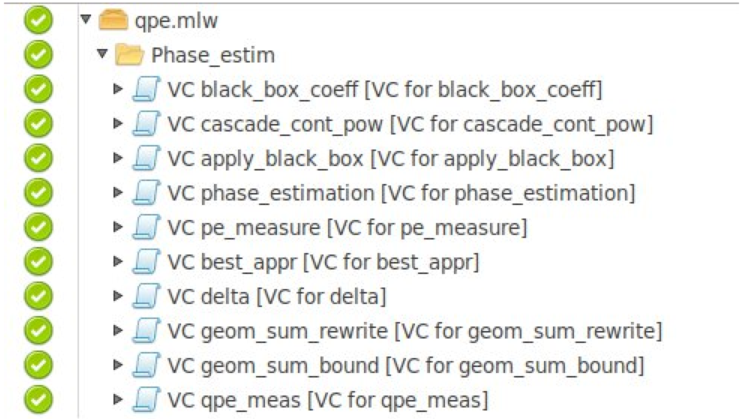
QBRICKS / Demonstration

Proof support

- **Interfaces**
  - calls to SMT-solvers
  - interactive proof commands, to help SMT-solvers

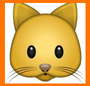
- **Output**
  - probation against complex case studies (× 6 vs SotA)
  - high-level (95%) of automation, proof effort × 1/3 vs SotA





Quantum memory states

- Classical world:

 **XOR** 

- Quantum world:

α_0  $\oplus$ α_1 

with $\alpha_0, \alpha_1 \in \mathbb{C}, |\alpha_0|^2 + |\alpha_1|^2 = 1$

Quantum memory states

- Classical world:

🐱 $_0$ 🐱 $_1$ ~~🐱~~ $_2$ 🐱 $_3 \cdots$ ~~🐱~~ $_{n-1}$

One sequence in $\{$ 🐱 , ~~🐱~~ $\}^n$ (over 2^n possible)

- Quantum world:

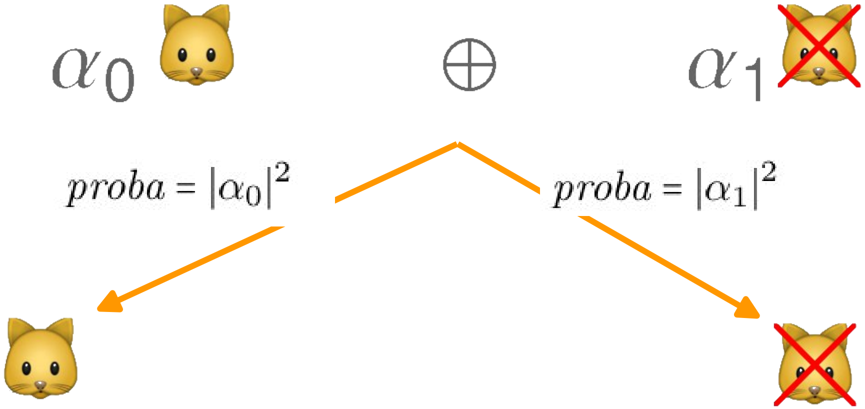
α_0 🐱 $_0$ 🐱 $_1$ 🐱 $_2$ 🐱 $_3 \cdots$ 🐱 $_{n-1}$

α_1 🐱 $_0$ 🐱 $_1$ 🐱 $_2$ 🐱 $_3 \cdots$ ~~🐱~~ $_{n-1}$

$\cdots$ $\cdots$

α_{2^n-1} ~~🐱~~ $_0$ ~~🐱~~ $_1$ ~~🐱~~ $_2$ ~~🐱~~ $_3 \cdots$ ~~🐱~~ $_{n-1}$

Quantum measurement



- Destructive
- Probabilistic

Quantum memory states

- Classical world:

+ some *strange* rules :

- **no cloning**
- **destructive measure**
- restricting set of operations (“unitary”)

. . .   . . .

α_{2^n-1} $_0$ $_1$ $_2$ $_3$ . . . $_{n-1}$

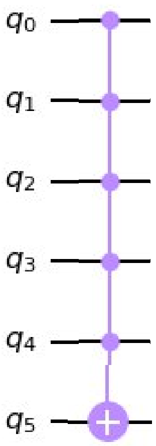
Formal verification and classical debugging : comparison

Testing/Assertion checking	Formal verification
tested instance	**any** instance
based on executions/simulations	**static** analysis, **no need to execute**
bounded parameters	**scale insensitive**
non deterministic programs : **statistical arguments**	absolute, **mathematical guarantee**

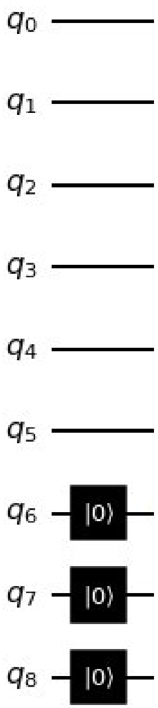
Build on **best practice** of formal verification for the classical case and **tailor them to the quantum** case

Bugs : ancilla qubits reallocation

Eg : Create a multiple control gate

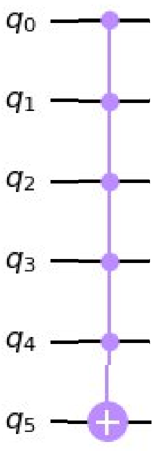


- add ancilla qbits

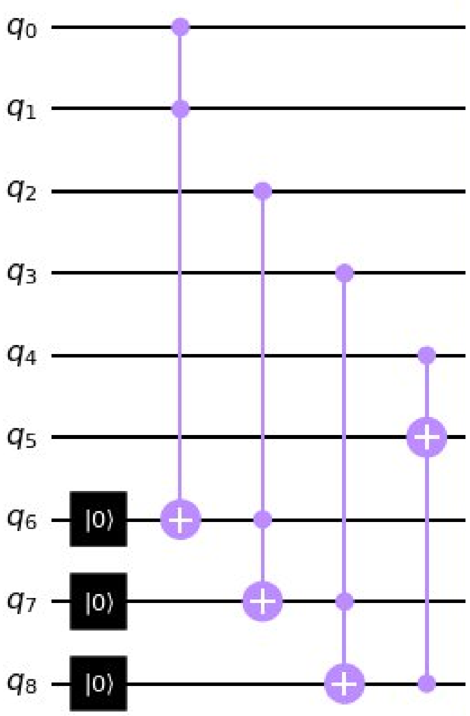


Bugs : ancilla qubits reallocation

Eg : Create a multiple control gate

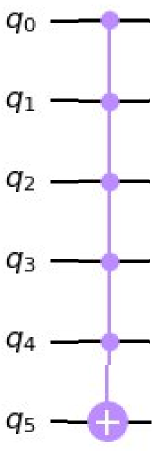


- add ancilla qbits
- use them to store control values

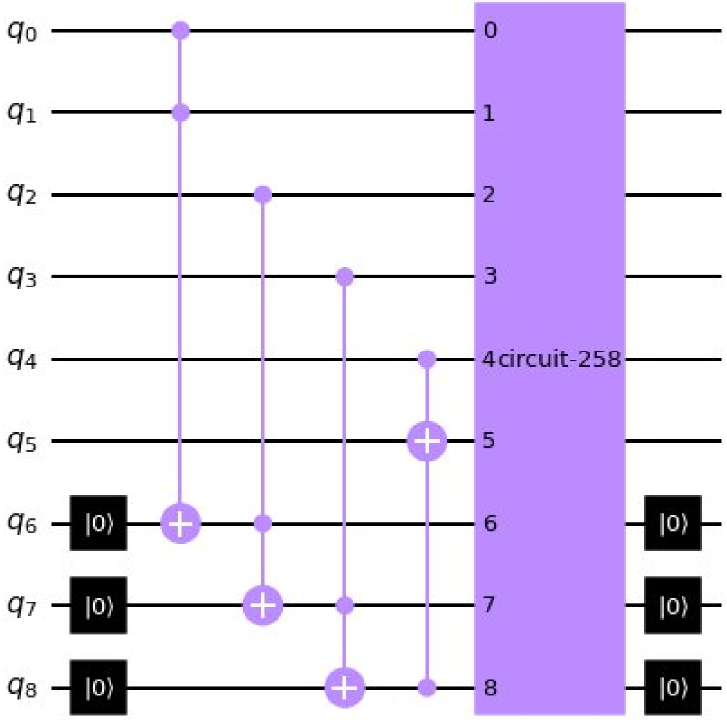


Bugs : ancilla qubits reallocation

Eg : Create a multiple control gate

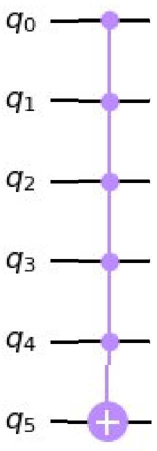


- add ancilla qbits
- use them to store control values
- free the ancilla qbits



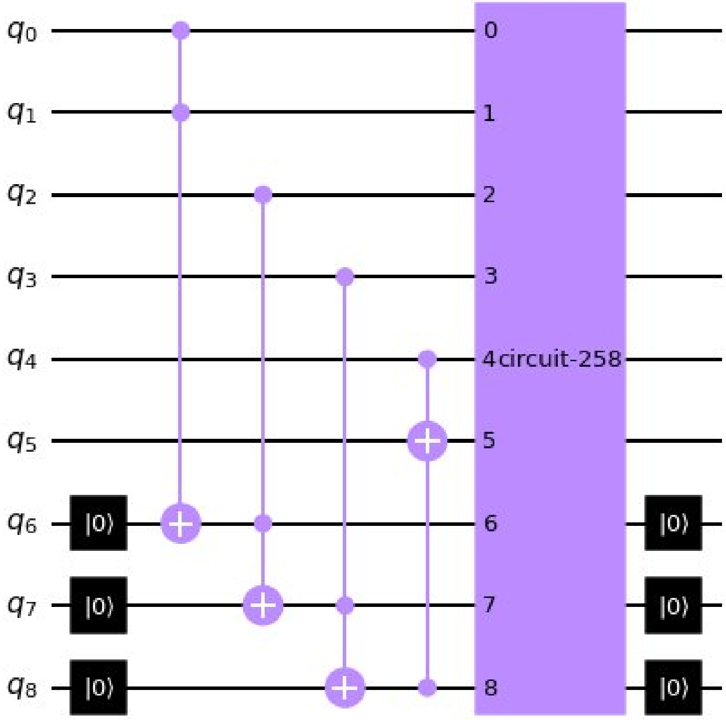
Bugs : ancilla qubits reallocation

Eg : Create a multiple control gate



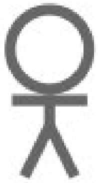
- add ancilla qbits
- use them to store control values
- free the ancilla qbits

requires{q[6-8] is uncomputed}



2022: towards a fully integrated verified programming environment → feedbacks

```
let qft ( n:int) :circuit
                    requires{0<n}
  =
  begin
    let c = ref (m_skip n)
       in for q = 0 to n-1 do
            invariant{width !c = n}
              invariant{range !c = q}
              invariant{forall x y i. 0<= i < n ->
                        basis_ket !c x y i = if 0<= i < q then y i else x i}
              invariant{forall x y. ang_ind !c x y = (ind_isum(fun k ->
                       (ind_isum (fun l -> x l * y k * power 2 (n-l - 1+k)) k n))0 q) /./ n}
              begin
                let cl = ref (m_skip n)
                   in for i = q+1 to n-1 do
                        invariant{width !cl = n}
                          invariant{range !cl = 0}
                          invariant{forall x y i. 0<= i < n ->
                                    basis_ket !cl x y i = x i}
                          invariant{forall x y. ang_ind !cl x y =
                                   (ind_isum (fun l -> x l * x q * power 2 (n- l -1+ q)) (q+1) i) /./n}
                          cl := !cl -- (crz i (q) (i - q+1) n );
                      done;
                      cl:=  place_hadamard (q) n -- !cl;
                      assert{forall x y i. 0<= i < n ->
                             basis_ket !cl x y i = if i = q then y 0 else x i};
                      assert{forall x y. ang_ind !cl x y =
                               (ind_isum (fun l -> x l * y 0 * power 2 (n-l - 1+ q)) q n) /./ n};
                      c:= !c -- !cl;
              end
          done;
          return (!c)
            ensures{width result = n}
            ensures{range result = n}
            ensures{forall x y i. 0<= i < n -> basis_ket result x y i = y i}
            ensures{forall x y. ang_ind result x y  = (ind_isum(fun k ->
                   (ind_isum (fun l -> x l * y k * power 2 (n-l - 1+k)) k n))0 n) /./ n}
  end
```



Pre-treatment (static analysis).

Automate

- Resource analysis
- Well-formedness (unitarity)
- functional verification

2022: towards a fully integrated verified programming environment → feedbacks

```
let qft ( n:int) :circuit
                   requires{0<n}
  =
  begin
    let c = ref (m_skip n)
       in for q = 0 to n-1 do
            invariant{width !c = n}
              invariant{range !c = q}
              invariant{forall x y i. 0<= i < n ->
                        basis_ket !c x y i = if 0<= i < q then y i else x i}
              invariant{forall x y. ang_ind !c x y = (ind_isum(fun k ->
                       (ind_isum (fun l -> x l * y k * power 2 (n-l - 1+k)) k n))0 q) /./ n}
              begin
                let cl = ref (m_skip n)
                   in for i = q+1 to n-1 do
                        invariant{width !cl = n}
                          invariant{range !cl = 0}
                          invariant{forall x y i. 0<= i < n ->
                                    basis_ket !cl x y i = x i}
                          invariant{forall x y. ang_ind !cl x y =
                                   (ind_isum (fun l -> x l * x q * power 2 (n- l -1+ q)) (q+1) i) /./n}
                          cl := !cl -- (crz i (q) (i - q+1) n );
                      done;
                      cl:=  place_hadamard (q) n -- !cl;
                      assert{forall x y i. 0<= i < n ->
                             basis_ket !cl x y i = if i = q then y 0 else x i};
                      assert{forall x y. ang_ind !cl x y =
                               (ind_isum (fun l -> x l * y 0 * power 2 (n-l - 1+ q)) q n) /./ n};
                      c:= !c -- !cl;
              end
          done;
          return (!c)
            ensures{width result = n}
            ensures{range result = n}
            ensures{forall x y i. 0<= i < n -> basis_ket result x y i = y i}
            ensures{forall x y. ang_ind result x y  = (ind_isum(fun k ->
                 (ind_isum (fun l -> x l * y k * power 2 (n-l - 1+k)) k n))0 n) /./ n}
  end
```

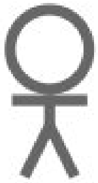
High-level programming:

→The case for **subcircuit control**

- **Common feature** in any reasonable implementation
- **Blind spot at every stage** in the dev/verif stack
  - user languages
  - formal analysis/semantics
  - compilation

Pre-treatment (static analysis).
Automate

- Resource analysis
- Well-formedness (unitarity)
- functional verification



2022: towards a fully integrated verified programming environment → feedbacks

```
let qft ( n:int) :circuit
                    requires{0<n}
  =
  begin
    let c = ref (m_skip n)
       in for q = 0 to n-1 do
            invariant{width !c = n}
              invariant{range !c = q}
              invariant{forall x y i. 0<= i < n ->
                        basis_ket !c x y i = if 0<= i < q then y i else x i}
              invariant{forall x y. ang_ind !c x y = (ind_isum(fun k ->
                        (ind_isum (fun l -> x l * y k * power 2 (n-l - 1+k)) k n))0 q) /./ n}
              begin
                let cl = ref (m_skip n)
                   in for i = q+1 to n-1 do
                        invariant{width !cl = n}
                          invariant{range !cl = 0}
                          invariant{forall x y i. 0<= i < n ->
                                    basis_ket !cl x y i = x i}
                          invariant{forall x y. ang_ind !cl x y =
                                   (ind_isum (fun l -> x l * x q * power 2 (n- l -1+ q)) (q+1) i) /./n}
                          cl := !cl -- (crz i (q) (i - q+1) n );
                      done;
                      cl:=  place_hadamard (q) n -- !cl;
                      assert{forall x y i. 0<= i < n ->
                                 basis_ket !cl x y i = if i = q then y 0 else x i};
                      assert{forall x y. ang_ind !cl x y =
                                (ind_isum (fun l -> x l * y 0 * power 2 (n-l - 1+ q)) q n) /./ n};
                      c:= !c -- !cl;
              end
          done;
          return (!c)
            ensures{width result = n}
            ensures{range result = n}
            ensures{forall x y i. 0<= i < n -> basis_ket result x y i = y i}
            ensures{forall x y. ang_ind result x y  = (ind_isum(fun k ->
                  (ind_isum (fun l -> x l * y k * power 2 (n-l - 1+k)) k n))0 n) /./ n}
  end
```

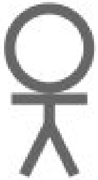
High-level programming:

→The case for **subcircuit control**

- **Common feature** in any reasonable implementation
- **Blind spot at every stage** in the dev/verif stack
  - user languages
  - formal analysis/semantics
  - compilation

Pre-treatment (static analysis).
Automate

- Resource analysis
- Well-formedness (unitarity)
- functional verification



Compilation : distribute over different

- **architectures**
- **computing models**

Providing **guarantees** and **analysis tools** :

- functional preservation
- resource estimations