Modeling the operation of Margolus quantum cellular automaton using high-level Petri nets

Elina Parviainen

May 2002

Master's Thesis

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

TEKNILLINEN KORKEAKOULU **DIPLOMITYÖN TIIVISTELMÄ**

Tekijä: Elina Parviainen

Työn nimi:
Modeling the operation of Margolus quantum cellular automaton using high-level Petri nets

Suomenkielinen nimi:
Margoluksen kvanttisoluautomaatin toiminnan mallintaminen korkean tason Petri-verkoilla

Päiväys: 6. toukokuuta, 2002	**Sivuja:** iv+64
Osasto: Tietotekniikka	**Professuuri:** T-119
Valvoja: Prof. Ilkka Niemelä	**Ohjaaja:** Emer. prof. Leo Ojala

Tiivistelmä:

Työn aiheena on tutkia Petri-verkkojen soveltuvuutta kvanttitietokoneiden mallinnukseen. Työssä esitetään yksi mahdollinen mallinnustapa ja arvioidaan sitä. Aluksi esitetään kvanttilaskennan peruskäsitteet sekä mallinnuksen kohteeksi valittu kvanttitietokonetyyppi, Margoluksen kvanttisoluautomaatti. Työssä rakennetaan Margoluksen kvanttisoluautomaatin toiminnalle malli, jonka avulla voidaan laskea automaatin aikakäyttäytyminen sekä mittausten vaikutukset siihen. Esimerkkinä käytetään 16 ohjausqubitista ja 16 dataqubitista koostuvaa syklistä automaattia. Mallinnus tehdään kahdessa osassa. Ensiksi kuvataan Petri-verkkona vastaavan klassisen soluautomaatin ohjausosan toiminta ja etsitään sen avulla ohjausosan saavutettavat tilat. Sen jälkeen rakennetaan Petri-verkkomalli, jolla voidaan laskea kvanttisoluautomaatin saavutettavien kantatilojen todennäköisyysamplitudit diskreetteinä ajanhetkinä. Tämän mallin toimivuutta tutkitaan laskemalla sama aikakäytös mallin avulla ja matemaattista sovellusohjelmistoa käyttäen. Mallin avulla pystytään suorittamaan haluttu laskenta, mutta varsin hitaasti verrattuna vastaavaan laskentaan matemaattisella työkalulla suoritettuna.

Avainsanat: Kvanttitietokoneet, Petri-verkot, Margoluksen soluautomaatti

HELSINKI UNIVERSITY OF TECHNOLOGY	**ABSTRACT OF THE MASTER'S THESIS**

Author: Elina Parviainen

Name of the Thesis:
Modeling the operation of Margolus quantum cellular automaton using high-level Petri nets

Name in Finnish:
Margoluksen kvanttisoluautomaatin toiminnan mallintaminen korkean tason Petri-verkoilla

Date: May 6, 2002	**Pages:** iv+64
Department: Computer Science and Engineering	**Chair:** T-119
Supervisor: Prof. Ilkka Niemelä	**Instructor:** Prof. Emer. Leo Ojala

Abstract:

The purpose of the work is to study the suitability of Petri nets for modeling of quantum computers. One possible modeling approach is presented and evaluated.
In the beginning the basic concepts of quantum computation are given and the target of modeling, the Margolus quantum cellular automaton, is introduced. The model which is built in this work can be used to compute the time behavior and effects of the measurements in the automaton. As a concrete example, a cyclic automaton consisting of 16 control qubits and 16 data qubits is studied.
The model is built in two parts. First, the corresponding classical automaton is modeled as a Petri net, which is used to find reachable states of the control part. Next, a Petri net model of the quantum automaton is built. It can be used to compute probability amplitudes of reachable basis states of the quantum cellular automaton at discrete moments of time. This model is evaluated by computing the same time behavior using the model and using a mathematical application program. It is possible to perform the desired computation using the model, but it is very slow compared to using a mathematical tool.

Keywords: Quantum computers, Petri nets, Margolus Cellular Automaton

Contents

List of symbols and notations

$\lvert\psi\rangle$	state vector
$\frac{\partial}{\partial t}$	partial derivative with respect to t
$\hbar$	Planck's constant
ln	natural logarithm
i	imaginary unit
U^{-1}	inverse of matrix
$U^{\dagger}$	conjugate transpose of matrix
$\sum$	sum
$\prod$	product
$\times$	Cartesian product
$\otimes$	tensor product
$x!$	factorial of x
$\subseteq$	subset
$\cup$	union
$\wedge$	logical and
$\vee$	logical or
$\neg$	logical not
$\exists$	exists
$[p]l$	l is in symbolic sum if p is true
Pr/T net	Predicate/Transition net
MCA	Margolus Cellular Automaton
MQCA	Margolus Quantum Cellular Automaton
PROD	Pr/T net reachability analysis tool
Matlab	Matrix-oriented tool for technical computing

Acknowledgements

I would like to thank professor Ilkka Niemelä for giving me the opportunity to carry out this master's thesis in the Laboratory for Theoretical Computer Science of Helsinki University of Technology. I am grateful to professor emeritus Leo Ojala for his valuable comments and advice. I would also like to thank my colleagues both in the laboratory and outside it, especially Harriet Beaver, Olli-Matti Penttinen and Teemu Tynjälä.

1 Introduction

1.1 Quantum computers

The idea that quantum mechanical systems could perform meaningful computations is about twenty years old. In 1980, Paul Benioff [1] formulated a model of Turing machine, where operations were done quantum mechanically. Although the model was not yet fully quantum, it managed to combine the mathematical model of computer with quantum mechanics. The relationship between classical and quantum systems was further clarified by Richard Feynman, who in 1982 showed [4] that a classical system, e.g. an ordinary computer, can not simulate a quantum system without an exponential slowdown. Later it was shown [11], [26] that although simulation of quantum systems by classical ones is inefficient, the simulation can be done effectively by a quantum computer.[3]

Peter Shor's factorization algorithm brought the quantum computers to general awareness. Shor showed in 1994 [20] that using a quantum computer it is possible to factor large integers in polynomial time, significantly faster than with any known classical algorithm. This gave reason to think that quantum computers might be able to perform computations more efficiently than classical computers. Another important breakthrough was Grover's quantum search algorithm in 1996 [9]. Using it a single database item can be find in square root of time needed by classical computer for the same task.

A quantum algorithm needs a quantum computer to run it. For the first real, working implementations we must probably wait for at least 20 years, but a couple of different quantum processor models have already been proposed. Benioff's original model has the serious downside that, in order to describe its behavior, all its computational paths have to be known – which effectively means that to solve a problem on Benioff's computer, we must already know the answer. Furthermore, the model exhibits quantum behavior only during a computational step. Between the steps it returns to a classical state.

A somewhat different model was developed by Feynman in [5]. His model is based on representing the operation of the computer as a kind of digital circuit. The circuit, along with some extra information to guide the progress of computation, is then systematically converted into representation of a quantum system.

In 1985, David Deutsch developed a fully quantum model of Turing machine. He also described a universal quantum computer, which could simulate any other quantum computer (or any other physical system) with arbitrarily high accuracy.

Feynman's ideas were generalized for a parallel computer by Norman Margolus, who introduced his model of a quantum cellular automaton in 1986 [12] and developed it further in [13]. This was the first time a quantum computer was made to mimic a classical parallel system. Later, also other quantum cellular automata have been developed [24].

1.2 Motivation

Petri nets [16] have proved to be a useful tool in modeling of many classical systems, including classical computers. The main goal of this work is to study, whether Petri nets would be a suitable tool for modeling a quantum processor as well. Margolus's cellular automaton was a natural choice for the target of modeling, because it is parallel (concurrent events are easily represented using Petri nets) and because there exists a simple and straightforward method for representing the automaton behavior as a quantum system [12],[13].

Petri nets have been earlier used for modeling a serial quantum processor by Ojala et al. in [15],[14]. In this work, we adopt the same approach as in [3], [15] and [14]: to compute the automaton state at successive discrete moments of time, based on the solutions of differential equations which describe the behavior of the automaton. The model in [15] and [14] hides the details of the computer evolution inside one transition. In this work we concentrate on showing the progress of computation in detail.

Our aim is to build a model, which could be used for computing the state of the automaton and studying the effect of measuring the state on the behavior of the automaton.

As our formalism we use the Predicate/Transition (Pr/T) nets. The behavior of quantum systems always contains a probabilistic element, which would lead thoughts towards stochastic formalism. As we intend to analyze the behavior of our system, rather than simulate it, the Pr/T net formalism contains all features we need in modeling. In addition, there are good analysis tools available for Pr/T nets.

1.3 Outline of the work

First, we introduce the basic concepts of quantum computing in Section 2, and the Predicate/Transition net formalism in Section 3. Then we describe the object of our modeling task, the Margolus cellular automaton, in Section 4.

Our model is built in two parts. First, the operation of classical version of the Margolus cellular automaton is modeled as a Petri net, and and its reachability graph is used to study which states are possible during the computation. The classical model, for an example automaton of 32 bits, and its reachability graph are built in Section 5.

Based on the results of the classical analysis we build a model for the operation of the quantum version of Margolus cellular automaton in Section 6. By performing the computations for reachable states only, it is possible to study substantially bigger quantum systems than would be possible by blindly using the equations for the full system.

The model can be used to compute the time evolution of the quantum version of the automaton. This is done and the result compared to the corresponding computation using Matlab in Section 7.

Finally, we present our conclusions in Section 8.

2 Basic concepts of quantum computing

In this section, we will cover briefly those basic concepts of quantum computing, which are necessary for understanding the structure and behavior of the quantum version of Margolus cellular automaton. For a more thorough treatment of the subject, see e.g. [10] or [18].

2.1 Single qubits and operations on them

Two-state quantum systems are called quantum bits, or shortly *qubits*. One possible way to implement them physically is to use so called spin-$\frac{1}{2}$ -particles, e.g. electrons. Mathematically, a qubit is represented as a column vector in a two-dimensional Hilbert space. In the scope of this work the qubits can be thought simply as vectors in a complex space with inner product defined; a more formally precise treatment of Hilbert spaces can be found e.g. in [10]. Quantum computing literature usually uses qubit $\binom{1}{0}$ for logical zero, and $\binom{0}{1}$ for logical one. In this work, following [12], the opposite convention is used: "spin up" vector $\binom{1}{0}$ denotes logical one, and logical zero is "spin down" vector or $\binom{0}{1}$. These two states are called the *basis states* of the one-qubit system.

A peculiarity of qubit states, compared to classical bits, is that they can be both in state 1 and in state 0 at the same time. A general one-qubit state can be represented as a linear combination, or a *superposition*, of the basis states:

$$\psi = \alpha \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \beta \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |\alpha|^2 + |\beta|^2 = 1, \tag{2.1}$$

where α and β are complex numbers. Coefficients α and β are called the *probability amplitudes.*

Qubit vectors can be manipulated by different *operators*. Operators are linear mappings which transform vectors into other vectors of the same (Hilbert) space. They are represented by matrices. The operators we will need for specifying the behavior of a quantum computer are *raising* and *lowering* operators (R and L respectively):

$$R = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad L = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}. \tag{2.2}$$

The raising operator produces $\binom{1}{0}$ if applied to qubit $\binom{0}{1}$, and lowering operator lowers a "spin up" , or $\binom{1}{0}$, into a "spin down" . If R is applied to a state which already is a "spin up" state, or L to a "spin down" state, the result will be a *null vector* $\binom{0}{0}$.

2.2 A system of several qubits

Many-qubit systems are represented as higher dimensional Hilbert space vectors, so that for k qubit system the dimension of the corresponding vector is 2^k. State vectors and operators for higher dimensional spaces are built from those of lower dimensional spaces using *tensor product*:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \dots & a_{1m}B \\ a_{21}B & a_{22}B & \dots & a_{2m}B \\ \vdots & \vdots & & \vdots \\ a_{n1}B & a_{n2}B & \dots & a_{nm}B \end{pmatrix}, \tag{2.3}$$

where B can be a matrix of any dimensions.

The basis vectors of a k-qubit system are formed by combining basis states for single-qubit systems using tensor product. For building a k-qubit state, either $\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ or $\left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$ state is chosen for each of the qubits, so that all 2^k combinations are used in some basis states of the k-qubit system.

The resulting vectors will have exactly one element with value one, and all other elements will be zeros. Instead of always writing the full vectors with all their zeros, we use so called *ket vectors* (a notation by Dirac). In ket vector notation, the one-qubit basis vectors are

$$|0\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \tag{2.4}$$

and vectors of higher dimensional spaces are

$$|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_k\rangle = |\psi_1\psi_2\dots\psi_k\rangle. \tag{2.5}$$

If we think of the one-dimensional basis vectors as numbers zero and one, then the higher dimensional vectors correspond to binary numbers (which, in turn, can e.g. represent a configuration of a computer). These binary numbers are written inside a ket vector, for example,

$$|00\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \quad |10\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}. \tag{2.6}$$

For representing state changes in k-qubit systems, we need operators which are able to operate on 2^k-dimensional vectors. Building such operators directly is tedious, so we would rather build the higher-dimensional operators using two-dimensional ones. This is possible by exploiting a useful property of tensor product (e.g. [17]):

$$\bigotimes_j (M_j \cdot v_j) = \bigotimes_j M_j \cdot \bigotimes_j v_j. \tag{2.7}$$

To put it simply, this says that we can convert a binary number (or the corresponding 2^k dimensional basis vector) into another by changing separately the necessary bits (two-dimensional basis vectors).

Using this property and the fact that lowering and raising operators transform states which they are not meant to handle into null vectors, we can build operators which operate on several qubits but "react" to some specific qubit combinations only. For other combinations, the result is a null vector (of a higher dimensional space). For example, an operator

$$R \otimes L \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes L \tag{2.8}$$

transforms a ket vector $|01x1\rangle$ into $|10x0\rangle$, (where x can have any value) and returns null for all other combinations. The example demonstrates also, how an operator can look for qubits in some specific positions only: identity matrices are used to operate on all "don't care" qubits. We can build operators reacting on several different qubit combinations by building an operator for each combination and summing them.

2.3 Entanglement

An operator usually takes a quantum state into some superposition of states. The resulting superposition seldom contains all possible basis states; instead, only some set of states, in a way or another meaningful for the computation performed, is present in the superposition.

Any single qubit of the state behaves as a superposition of $|0\rangle$ and $|1\rangle$. If, however, we are able to exactly determine the value of some of the qubits, it may be possible to infer also the values of other qubits exactly, because only certain combinations were present in the original state. For example, if we know that the first qubit of state

$$|\psi\rangle = \alpha|01\rangle + \beta|10\rangle, \tag{2.9}$$

is in state $|0\rangle$, we can be sure that the second qubit is in state $|1\rangle$. There are connections, so called *entanglement*, between qubits, which can not be noticed when looking at the qubits separately. The state of the system as a whole contains more information than can be stored in the corresponding number of single qubits.

Because of entanglement, when we are studying the behavior of a quantum computer, we must look at the state of the whole system instead of separate qubits. The number of states growing exponentially in the number of qubits makes it difficult to study quantum systems consisting of more than a few qubits.

2.4 Measurements

A quantum system may be in a superposition of several states at once when its left undisturbed, but when we measure a quantum state, we get only one value as result. Furthermore, the measurement causes a superposed state to *collapse* into one of the basis states of the system. (This is the view on measurements according to so called Copenhagen interpretation of quantum theory; there are other interpretations as well [10]).

We can measure the state of a system using different *observables*. An observable is some physical property of a quantum system, e.g. the direction of the particle spin measured along some axis. Using different observables corresponds to representing the state vector in different bases. For example, in the case of the spin-$\frac{1}{2}$ particles, measurement along z-axis corresponds to representing the particle state in the basis {"spin up along z-axis", "spin down along z-axis"}.

An observable is represented by a matrix. The eigenvectors of the matrix are the basis states of the system, and the eigenvalues are the possible outcomes of the measurements.

When we measure the system state, the state collapses into the eigenvector which corresponds to the eigenvalue we got as measurement result. If the system consists of several qubits, only some of which are measured, then the state collapses into a superposition of basis states consistent with the measurement result.

If the system state vector happens to be one of the eigenvectors of the observable, measurement always gives as result the eigenvalue corresponding to that eigenvector. Generally, however, the state is a superposition, with some probability amplitude α_i assigned to each of the basis vectors ψ_i. The measurement outcome is still one of the eigenvalues λ_i, but the choice between different outcomes is probabilistic. The *probability* of measuring the eigenvalue λ_i (and thus collapsing the state into ψ_i) is $|\alpha_i|^2$. The coefficients α_i are normalized so that the sum of the probabilities is 1.

If the observable is not mentioned, it is assumed that some standard observable, e.g. a measurement along z-axis for spin-$\frac{1}{2}$-particles, is used.

2.5 Evolution of a closed system

All quantum systems we meet in this work are *closed systems*, that is, it is assumed that, except for the measurements, the systems have no interactions with the outside world. The behavior of a closed quantum system is completely described by Schrödingers's differential equation

$$\frac{\partial|\psi(t)\rangle}{\partial t} = -\frac{\mathrm{i}}{\hbar}H(t)|\psi(t)\rangle, \tag{2.10}$$

where $|\psi(t)\rangle$ is the state of the system at time t, $\hbar$ is Planck's constant and $H(t)$ is a Hermitian matrix called *Hamiltonian.* To simplify calculations, we will omit the constant $\hbar$, the only effect of this being a change in the time scale used.

The vector $|\psi(t)\rangle$ is called the *state vector* of the system. It is a superposition of basis states of the system, and its elements are the probability amplitudes of different basis states at the moment t.

The evolution given by (2.10) is only disturbed by measurements, which cause the system state to fully or partially collapse. It's not possible to ask the quantum system to stop its evolution until we are ready with the measurement. We assume, however, that all the measurements are made in a very short time, so that the evolution of the system during the measurement can be neglected. That is, we assume that after the measurement, the parts of the system which were measured (or because of entanglement affected by the measurement) are in the collapsed state, and any unmeasured parts are in the same state as immediately before the measurement.

If the Hamiltonian is time-independent, $H(t) = H$, the solution of (2.10) is

$$|\psi(t)\rangle = U(t)|\psi(0)\rangle, \tag{2.11}$$

where the evolution operator $U(t)$ is computed as a matrix exponential of H,

$$U(t) = \mathrm{e}^{-\mathrm{i}Ht} = \sum_{k=0}^{\infty} \frac{(-\mathrm{i}t)^k}{k!} H^k. \tag{2.12}$$

As we are going to study the behavior of the system at discrete time steps, it is useful to represent $U(t)$ as a power of operator U_{step}, which represents the evolution during one time step of length Δt:

$$U(t) = U_{step}^{\frac{t}{\Delta t}}, \qquad U_{step} = e^{-\mathrm{i}H\Delta t}. \tag{2.13}$$

Now,

$$U_{step}|\psi(t)\rangle = |\psi(t + \Delta t)\rangle. \tag{2.14}$$

This is (an exact) difference equation for the system evolution.

As each multiplication by U_{step} takes the system one time step forward in time, we can as well use as variable the step number instead of time,

$$U_{step}|\psi(n)\rangle = |\psi(n+1)\rangle, \quad t = n \cdot \Delta t. \tag{2.15}$$

From now on, all $|\psi(number)\rangle$ mean $|\psi(n)\rangle$, unless the variable t is explicitly mentioned.

The role of Hamiltonian in Schrödinger's equation (2.10) is to tell, which kind of connections there are between different basis states of the computer. As the laws of quantum physics are reversible, also any description of a quantum computer must be reversible in time [5]. Reversibility is ensured by demanding that the Hamiltonian is a Hermitian matrix. This results in the evolution operator in (2.12) being unitary (the operator can be inverted simply by taking a transpose conjugate).

When we speak of reversibility, thinking about classical systems, we usually mean that the system moves from state to state, and each state has a unique predecessor state; it's always possible to say "where the system came from".

A state of a quantum system, on the other hand, may consist of several basis (classical) states at the same time, with different probability amplitudes. Some of these states may be successors or predecessors of each other, in the sense that the Hamiltonian dictates that it is possible to move between them. When we say that a quantum system is reversible, what we mean is that if

$$U_{step}|\psi(n)\rangle = |\psi(n+1)\rangle, \tag{2.16}$$

then

$$U_{step}^{-1}|\psi(n+1)\rangle = |\psi(n)\rangle, \tag{2.17}$$

that is, by taking a step backwards the system returns to that superposition of states which preceded the previous forward step, instead of simply moving from a basis state into another.

2.6 Designing a quantum processor

A quantum processor can be designed by describing the evolution operator directly, or by building a Hamiltonian which, when exponentiated, will realize the desired evolution. Gramß discusses these approaches in [8].

A straightforward way to describe the behavior of a quantum computer is to build its time evolution operator U to represent the operation we want to perform during one time step (e.g. a boolean function of some reversible gate). U alone would be enough for studying the behavior of the processor, but for physical realization, it's always necessary to know the Hamiltonian H. It can be calculated from U by diagonalization and matrix multiplication, but as the matrix size grows exponentially in the number of qubits ($2^k \times 2^k$ for a k-state computer), this requires a lot of work. An evolution operator built this way describes the behavior of the whole computer. Following the traditions of the classical computer designs, it would seem a good idea to divide the computer into locally interconnected subsystems and describe the behavior of the whole system based on smaller building blocks.

If we build the system from interacting subsystems $U = \prod U_i$, it is usually the case that the order in which the subsystems operate is important, that is, the operators U_i don't commute. It is easy to calculate the Hamiltonian for each small subsystem. If the subsystems were noninteracting, the Hamiltonian of the whole system could be simply calculated as the sum of the local Hamiltonians H_i. When the subsystems interact, however, this kind of Hamiltonian does not take care of the correct operating order of the subsystems.

There are two known solutions to this problem. First, the Hamiltonian can be made time-dependent. That means that either the operator of the computer or some classical system changes the energy couplings in the quantum computer, so that first H_0 is realized, thereafter H_1, H_2 and so on, in correct order. This will cause heat dissipation during the computation, considerably slow the computation down, and may not always be possible to realize physically.

The second solution is to add to the system some extra qubits, and use these to add into the Hamiltonian some information about the correct order of U_i. This method was introduced by Feynman in [5], and later generalized by Margolus for parallel computers in [12, 13]. We will review Feynman's idea shortly in Section 4.3, and then explain Margolus's method in detail.

3 Predicate/Transition net formalism

The formalism we use is that of Predicate/Transition nets, defined in [7].

A Pr/T net $N_L = (N, A, M_0)$ consists of an underlying Petri net N, its annotation A and an initial marking M_0.

The underlying net N describes what places and transitions N_L has, and how they are connected with each other via arcs. Graphically, places are denoted as circles, transitions as rectangles and arcs as arrows between them.

Definition 1 (Underlying net)
N is a directed net, $N = (S, T; F)$.

- S is a set of *places*.
- T a set of *transitions*;
- $F \subseteq (S \times T) \cup (T \times S)$ is a set of *arcs*. If the arrow goes from (to) a place into (from) a transition, the place is called an *input (output) place* of the transition, and the arc an *input (output) arc*.

For annotating a net we use a first-order language L, which consists of *terms* and *formulae*. In addition, we define *symbolic sums* as a compact way of representing a set of tokens. Usually, a place is allowed to hold multiple similar tokens. Our model, however, will built so that it is enough to use *strict* Pr/T formalism, in which only one copy of each token is allowed inside a place.

Definition 2 (Terms and formulae)
Exactly the following are *terms*:

- variables,
- expressions $f(v_1, \ldots, v_n)$, where $f^{(n)}$ is an n-ary operator and v_i are terms. 0-ry operators are used as names of distinct individuals.

Exactly the following are *formulae*:

- $v_1 = v_2$, where v_1 and v_2 are terms,
- $Pv_1 \ldots v_n$, where $P^{(n)}$ is an n-ary predicate and v_1 and v_2 are terms,
- $\neg p_1$ and $(p_1 \vee p_2)$, where p_1 and p_2 are formulae,
- $(\exists x)p$, where x is a variable and p is a formula.

Definition 3 (Symbolic sum)
Exactly the following expressions are in the class of symbolic sums of tuples of length n:

- the constant $\mathbf{0}$ (a combination where all coefficients are 0)
- n-tuple $\langle v_1, \ldots, v_n \rangle$, where v_i are terms,
- $(l_1 + l_2)$, where l_1 and l_2 are symbolic sums,

- zl, where l is a symbolic sum. z is either 0 or 1 (this restriction is due to the use of strict Pr/T nets)
- $[p]l$, where p is a formula and l is a symbolic sum.

The last item of the definition above is a minor extension to the basic form of Pr/T nets [7]: l is interpreted to be part of the symbolic sum only, if p is true. By using this extension we can in many occasions use one transition instead of many almost-similar transitions.

Now we are ready to give a definition for annotation A. The annotation gives information about what kinds of tokens the places can hold, which values are allowed inside tokens and how transitions behave.

Definition 4 (Annotation)
$A = (A_N, A_S, A_T, A_F)$, is an *annotation* of N, in a first-order language L.

- A_N is a first-order structure of L, called the *support* of N_L. In this work, complex numbers (our *carrier set*) and operations on them form A_N.
- A_S is a mapping from the set of places S to the *arities* of the places. Arity tells the size of the tuples which a place can hold in a marking.
- A_T is a mapping from the set of transitions T to a set of formulae, called *firing conditions*.
- A_F is a mapping from the set of arcs F to the set of *arc expressions*, which are symbolic sums of tuples of terms in L. The sizes of the tuples in the sum must match the arity of the place where the arc is connected.

M_0 is the *initial marking* of the net. It assigns to each place a symbolic sum of tuples, which contain elements of the carrier set. These tuples are called *tokens*. The arity of the tokens must match the arity of the place.

A marking of Pr/T net changes into another marking, when transitions *fire*. If the firing condition holds, output places don't previously contain tokens which the transition would produce into them, and all the variables on the input arcs of a transition can be substituted by values taken from the input places, the transition is *enabled* in the current marking. An enabled transition can fire, yielding another marking. In the new marking, tokens corresponding to expressions on input arcs have been removed from the input places, and into output places, tokens corresponding to output arc expressions have been produced. In the graphical notation, firing conditions are written inside the transitions and arc expressions near the corresponding arcs. Markings are written inside the places.

4 Margolus Cellular Automaton

4.1 Classical version of the automaton

Cellular automata were originally introduced by von Neumann [23] for studying self-reproduction in biological systems. Since then, they have been used for various purposes, e.g. formal language recognition and modeling of physical systems. A new era in cellular automata research was started by Wolfram [25], who shifted the focus from applications to study of mathematical properties, e.g. statistical parameters, entropy and topology of automata.[19]

An introduction to the subject is found e.g. in [21]. Readers wishing to have an overview on types and use of cellular automata are referred to [19].

Cellular automaton consists of several units, called *cells*, which change their state according to some set of rules. All cells are identical in the sense that they follow the same rules. On each time step, the current state of some *neighborhood* (a set of cells which can but does not need to contain the cell itself), determines the new state of the cell. A cellular automaton can be of any dimension, one- and two-dimensional being the most commonly used. In this work, we only come to contact with two-dimensional automata. Its cells are thought to be located on a grid, usually of a finite size. The automaton can be *cyclic* or *acyclic*. In the cyclic variant, the cell grid is finite, and is thought to behave like a torus, where the cells on the left edge have the cells of the right edge as their neighbors, and similarly for the upper and lower edges. In a finite acyclic automaton, the cells on edges of the grid of cells must be handled as special cases, as they may not have a full neighborhood. If the grid is infinite, any special cases are not needed.

In [21], Toffoli and Margolus introduce a bit different variant of cellular automaton, called *partitioning cellular automaton.*

In partitioning automaton, the grid of cells is partitioned into disjoint *blocks*, and the evolution rules are given for a block, not for a single cell. A block rule gives the new state of a block based on the old state of the block, without any reference to the neighboring cells or blocks. From one time step to the next, the partitioning is changed so that blocks of one step overlap with those of the next step. This creates interaction between blocks and turns the automaton into a dynamical system. This kind of automaton is useful, especially when modeling microscopically reversible systems, because reversibility of the whole system can be ensured simply by making the block rules reversible. A block rule is reversible, if there is a one-to-one correspondence between the initial and final states of a block.

Margolus cellular automaton (MCA) is a simple special case of a partitioning automaton. The grid is divided into disjoint blocks, each consisting of 2×2 cells. Partitioning changes so that on odd steps the upper left corner of a block is located on odd coordinates (these blocks are called *odd blocks*), on even steps it's located on on even coordinates (the *even blocks*)). The division is illustrated along with an example evolution in Figure 1.

4.2 Local synchronization

When we speak of cellular automata, we have usually in mind a globally synchronized automaton. There all the cells change their values at the same time. Implementing this is not a big problem when the automaton is simulated on an ordinary computer. On

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

0	0	1	1	1	1
0	0	1	1	1	1
0	0	1	1	1	1
0	0	1	1	1	1
0	0	0	0	0	0
0	0	0	0	0	0

0	0	1	1	1	1
0	0	1	2	2	1
0	0	1	2	2	1
0	0	1	1	1	1
0	0	0	0	0	0
0	0	0	0	0	0

0	0	1	1	1	1
0	0	1	2	2	1
1	1	1	2	2	1
1	1	1	1	1	1
1	1	1	1	0	0
1	1	1	1	0	0

0	0	1	1	1	1
0	0	1	2	2	1
1	1	1	2	2	1
1	2	2	1	1	1
1	2	2	1	0	0
1	1	1	1	0	0

Figure 1: An example of possible evolution in an asynchronous automaton. The numbers show cell ages, and the division into blocks on each round is shown with thick lines. On odd rounds, any odd block with even ages can update, and on even rounds the even blocks can update if the block cells have odd ages.

a quantum system, however, making all the qubits behave synchronously would require explicitly taking into account the connections between even the farthermost qubits. We would no longer be able to build the automaton from local subsystems. To describe the state changes of a globally synchronized quantum automaton, we would need to fully analyze the possible configurations and moves between them. In effect, we would need to perform the computation beforehand in order to build a computer to perform it.

It is possible, however, to perform an effectively synchronous computation using only local synchronization [13]. The trick is to use extra variables to keep track of the computational age (the number of performed updates) of the cells. Each time a block is updated, the ages of all its cells increase. The new cell values of a block on the kth step can be computed only when all its cells have had their values updated on the $(k-1)$th step. Because the division into blocks is different on even and odd steps, an even (odd) block must wait for all the odd (even) blocks partially overlapping with it before it can proceed. Those blocks, in turn, may be waiting for blocks a bit farther away, and so on.

As the result, some cells are older and some younger, but the differences between adjacent cells are always less than or equal to one time unit (see Figure 1 for an example). The operation of a locally synchronized automaton can be compared to a chain of people walking hand in hand: a person can take a step forward only if his both neighbors are following him near enough.

Local synchronization is implemented in MCA by keeping the original cells and their values (the *data bits* of the automaton) intact, but adding to each cell an extra bit, a *guard bit*. As said above, the function of this extra variable is to measure the age of the cell. But, using the given updating scheme, ages of adjacent cells can differ at most by one time unit, and thus it is enough to use only one bit instead of a step number. All guard bits are initially set to zero. The step numbering starts from 1. On odd steps the block is allowed to update, if all its guard bits have value zero, meaning "the cell has been updated an even number of times, now the odd step can be done". On even steps, a block with "1" in all guard bits can update. After updating the values of the data cells of a block, also it's guard bits are flipped.

As our goal is a quantum version of this locally synchronized automaton, we must enable also backward operation, to make the automaton reversible. The behavior of the data part can be made reversible simply by using reversible block rules. In reverse operation of the guard bit system, the roles of "1"s and "0"s are reversed. Now odd steps react to a block of guard bits with value "1", meaning "the next, even, step is complete and its effect must now be canceled to produce the situation on this, odd, step". Similarly, on even steps a block of zero-valued guard bits is updated, and the guard bits flipped. On each step, the automaton can operate either forwards or backwards.

4.3 Quantum MCA (MQCA)

Now when we are familiar with the classical version of the MCA, we would like to build a quantum system, which mimics its behavior. Margolus gives a method for doing this in [12],[13]. We restrict ourselves into studying two-dimensional cyclic automata with binary valued data cells, the same kind that Margolus describes in his works.

The one bit classical variables can be naturally replaced by qubits in the quantum automaton. Possible configurations of the classical automaton become the basis vectors of the quantum system. The evolution from state to state is described by a suitably built Hamiltonian, as described in the following subsections.

4.3.1 Feynman's computer

Margolus's ideas of quantum cellular automaton are based on a simpler, serial quantum processor, which Feynman presents in [5]. Starting point is a kind of logical circuit, a series of gates (unitary operators) U_{Fi}, which should operate in a certain order to produce the final result. The overall operation of Feynman's computer is thus described by a unitary matrix

$$U_F = U_{Fk} \cdot \ U_{Fk-1} \cdots \ U_{F2} \cdot \ U_{F1}. \tag{4.1}$$

The operation of a quantum computer, however, is not directly guided by this unitary operator, but by a corresponding Hamiltonian H_F. It would be nice to build the computer from small components U_{Fi} without computing full U_F for every circuit. However, finding the necessary H_F using only properties of matrices U_{Fi} is difficult.

Feynman noticed that the total state of the computer is produced as a superposition of states, where the Hamiltonian has operated on the initial state an arbitrary number of times, as can be seen by expanding $U_F = e^{-iH_F t}$ as given in (2.12). When measured, such a superposition can collapse to any of the states which formed the superposition, and the observer has no way of knowing whether the value measured was the final result, or some situation where only some of the gates U_{Fi} have operated. To make it possible to see which situation has been measured, Feynman made each state in the superposition consist not only of the state of data qubits the computer is operating on, but also a number of so called program counter qubits.

The program counter qubits indicate how many gates U_{Fi} have operated to produce the situation present at the data bits. For a circuit of k gates there are $k+1$ program counter qubits. Initially, first of them is in state $|1\rangle$, and all others in state $|0\rangle$, to indicate that no gate has operated yet. Each time a gate operates, the $|1\rangle$ is moved to the next counter qubit. If a gate operates backwards, as reversible gates of a quantum computer can do, the $|1\rangle$ is moved to the previous counter position. Feynman gave a method of building the Hamiltonian of matrices U_{Fi} and creation and annihilation operators (which correspond to lowering and raising operators, respectively) to implement this behavior. Margolus's way of building the cellular automaton Hamiltonian, presented in the following, is a generalization of this method for a parallel computer.

4.3.2 Hamiltonian of MQCA

The Hamiltonian H_M for the two-dimensional version of MQCA is formed as a sum of terms which describe the behavior of different blocks (see [12]):

$$H_M = \sum_{X,Y} (F_{XY} + F_{XY}^\dagger). \tag{4.2}$$

Each term F_{XY} describes the evolution of a block (whose upper left corner is at (X,Y)) forward in time, while its conjugate transpose $F_{XY}^\dagger$ describes the backward step of the same block. Block behavior is computed using raising operators $R_{x,y}$ for individual guard qubits for cells (x,y), the corresponding lowering operators $L_{x,y}$, and matrices A_{XY} which implement the actual block rules.

$$F_{XY} = \begin{cases} A_{XY} R_{x,y} R_{x+1,y} R_{x,y+1} R_{x+1,y+1}, & \text{block } (X,Y) \text{ even} \\ A_{XY} L_{x,y} L_{x+1,y} L_{x,y+1} L_{x+1,y+1}, & \text{block } (X,Y) \text{ odd} \end{cases}. \tag{4.3}$$

The operators for a given block are formed by using raising and lowering operators to handle the cells in the block, and operating by identity matrices on other cells, as explained in Section 2.2.

Note that $F_{XY}^\dagger$ is F_{XY} with "even" and "odd" reversed. Therefore, for each block (X,Y) there will be one unconjugated and one conjugated term, and thus in (4.2) we don't need to care about blocks being even or odd.

The product of the four raising operators R_{xy} in (4.3) reacts on a block of four guard qubits in state $|0\rangle$ producing a block of $|1\rangle$-states. If even one of the locations $(x,y), (x+1,y), (x,y+1), (x+1,y+1)$ happens to contain a $|1\rangle$, the raising operator applied to it gives a null vector, and the whole term F_{XY} goes to zero and has no effect in Hamiltonian. The combination of the four lowering-operators does the same, but with the roles of $|1\rangle$- and $|0\rangle$-states reversed. This way, the matrix A_{XY}, which takes care of the handling of data qubits, can operate only when the guard qubits of the (x,y) block have the right combination of values.

Matrices A_{XY} implement the actual rules which determine the behavior of data blocks. They are built using three kinds of operators: lowering and raising operators to check that a cell has a certain value and changing it into another, and projection operators (which check that a cell has a certain value without changing it). If the rules used are reversible, the resulting matrices are unitary. As the cells of the automaton are similar, all the matrices A_{XY} will implement the same behavior, the only difference between them being that they operate on different blocks.

By looking at Schrödinger's equation (2.10) we notice that the Hamiltonian describes how the state of the automaton behaves in an infinitesimally small time step. Effectively, the Hamiltonian encodes the information about which configurations (basis states) can be reached from which taking only one step between the configurations. We call the steps between basis states *logical steps*, to make a distinction between them and the *time steps* used in computation (Δt). If several logical steps are taken (i.e. the Hamiltonian is exponentiated, like in (2.12)) some bigger set of states can be reached from the initial state.

The basis states which are this way reachable from the initial state are called *computational states* of the automaton. Because a Hermitian matrix always describes reversible behavior, any of the computational states can be reached from any computational state, if an arbitrary number of steps can be taken.

Because all possible ways of moving between configurations (all possible powers of H_M) are present in the time evolution operator U_M, the automaton does not evolve from configuration to configuration, as is the case in the classical automaton, but U_M changes any initial (computational) state into a superposition of all computational states. The relative importance of each computational state in the superposition, the probability amplitude, is determined by the initial state and the length of the time the automaton has evolved. Thus the probability of finding the automaton in a certain configuration when measured varies over time.

4.3.3 Initial state of the automaton

In a classical, locally synchronized cellular automaton, all the guard bits were initially set to zero. This is a suitable approach when the computer is always computing forwards. For the quantum computer, however, there is no difference between a step forward and a step backward; both are equally likely. Let's assume that we set all guard qubits initially to $|0\rangle$. In the first step, any odd block can take a forward step and any even block can go backwards; there are the same number of "ready to go forward" and "ready to go backward" blocks. However, already after updating one block, this is no more true. If, for example, one odd block takes a step forward, it prevents all the even blocks partially overlapping with it from changing, and as a result, there are more blocks which can go forward than those which can go backward. Other imbalanced situations are created when more blocks are updated.

As long as the number of "ready to go forward" blocks equals that of "ready to go backward" blocks, the computation can be said to progress with constant velocity. (The concept of computational velocity in MQCA is somewhat off the topic of this work; readers interested in more detail are referred to [13] and [2]). Therefore, it would be desirable to maintain a balance between the number of blocks ready to take steps in different directions.

To make sure that the number of "ready to go forward" and "ready to go backward" blocks is always the same, Margolus uses a special initial state of the guard bits, as shown in Figure 2. This configuration allows only two (overlapping) columns of blocks to be active at a time. When enough blocks on one columns are ready, some blocks of the next are updated, and so on, until, after updating the last column, the update cycle starts from the first column again. This method restricts the two-dimensional automaton to operate parallelly in one direction only: on a column, several blocks can be ready for update at once, but in each row of a configuration, there is exactly one active region at the time.

4.3.4 Measuring the result

In Feynman's computer the relationship of the measured state of the cursor system and the state of result qubits is simple: when last of the cursor qubits is $|1\rangle$, the computation is ready. If not, the evolution is let to continue. In MQCA the situation is similar, but not quite that simple. Also here the automaton state is a superposition of valid computational states. Which of them can be considered "final" states is not determined by the working

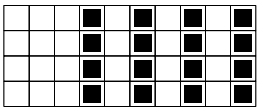


Figure 2: The special initial state of Margolus quantum automaton. In the initial situation, only the blocks in columns 1 and 2 (or 2 and 3) have all guard qubits in the same state and thus can be updated.

order of quantum gates, as is the case in Feynman's computer, but must be decided based on the computation. Therefore, repeatedly measuring the guard bits to find some predefined combination may be a suitable way of finding the result only in some special cases.

Classical cellular automata are often used for different simulations, where there is no clear division between intermediate states and final results. The only "result" is the sequence of states the automaton goes through. This kind of approach is clearly not suitable for quantum automata.The states of the quantum automaton are superpositions, and thus we cannot say that "at the moment t the configuration of the automaton is c". Also, we cannot see any individual configurations without destroying the superposition.

Instead, quantum automata might be suitable for simulating computations, where the cells and rules of the automaton form some kind of computing device, e.g. by simulating the operation of a digital circuit. In this type of automaton, the computation could be arranged so that when a certain bit combination, "a computation done flag", appears in certain place on the data grid, we can be sure that some other data bits contain the result [12]. Thus a suitable measuring scheme for MQCA is to repeatedly measure some data bits (partially collapsing the automaton state) and measure the final result if the first measurement revealed right values. Otherwise, the computation is let to continue.

4.4 An example of MQCA

As a concrete example, we will study a cyclic 4×4 automaton, with one grid of data qubits and one grid of guard qubits.

The cells of the grids are numbered, so that each qubit has its own unique number. This numbering determines, in which order the qubit variables are used when the state of the whole system is built from them using tensor product. The numbering also determines, in which order the qubit values are written when the data configuration is represented as a string of ones and zeros. The numbering can be chosen arbitrarily. We number the qubits from left to right and from top to bottom. For example, the state of guard qubits in Figure 3 is represented as vector $|0000000000001111\rangle$ and that of the data qubits as $|1000000000000001\rangle$.

Only 16 data qubits are too little for building imitations of digital circuitry or other complex computations which a real quantum automaton could be thought to perform. Therefore, we will use as our example "computation" a simple system of two "particles", represented by qubits with value $|1\rangle$, moving on a background of $|0\rangle$ qubits towards each other. The particle movements are determined by so called Billiard Ball Model [6] rules (Figure 4), a rule set used also by Margolus in [12]. The rules are reversible and preserve the number of $|1\rangle$'s.

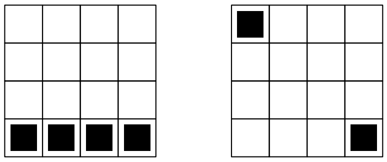


Figure 3: The initial state of the example automaton. Guard qubits are on the left and data qubits on the right.

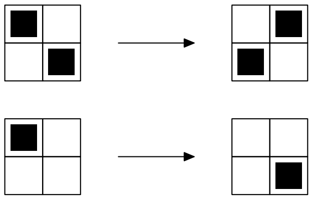


Figure 4: The Billiard Ball rules. The upper rule represents a situation where two "particles" in the same block "collide", and continue to the direction perpendicular to the original. In the lower rule a single "particle" moves forward, from a corner of a block to another corner. All situations with one "particle" or two "particles" in opposite corners behave symmetrically to those depicted here. In other cases, the block contents don't change.

Initially, the particles are located in the opposite corners of the qubit grid. The initial values of the guard bits are set to the special initial state used by Margolus: the three upper rows are filled with $|0\rangle$s and the lowest row with $|1\rangle$s. Figure 3 shows this initial state.

The measurement scheme we have chosen is somewhat artificial, but illustrates the "computation done -flag" technique described in Section 4.3.4. We study repeatedly the position of one of the "particles". If it is found in cell 6, near the middle of the data grid, we decide it is time to find out where the other particle is, and measure the whole state of the automaton (both guard qubits and data qubits).

5 Reachability of classical MCA states

5.1 Classical MCA with guard bits

In this section we concentrate on studying the guard bit system of the classical, locally synchronized automaton. Its guard bits determine how the computation can proceed, and the changes of the data bits just follow the changes of the guard bits. We'll build a net model describing the behavior of the classical MCA, and use it for building the Hamiltonian for the quantum automaton. This way, we don't need to use huge matrices, as would be the case had we built the Hamiltonian using (4.2) and (4.3) directly.

Effectively, the Hamiltonian tells the quantum automaton in what possible ways a corresponding classical automaton can behave. The quantum automaton has the same set of (basis) states and the same connections between them as the classical one, but it uses them in a quantum way: instead of being in one state at a time, it can be in a superposition of all possible states at once. To understand how the information of classical behavior is encoded for the quantum automaton to use, we have to first take a look at the structure of the MQCA Hamiltonian.

Each column in the Hamiltonian tells into which basis states some basis state can evolve during one logical step. The leftmost column corresponds to the basis state vector $\left(\begin{smallmatrix} 1 & 0 & 0 & \cdots & 0 \end{smallmatrix} \right) = |11\cdots 11\rangle$, and the other columns from left to right correspond to the ket vectors with binary numbers in descending order.

In the equation (4.3) the automaton behavior was described in terms of big matrices, which operate directly on the guard qubit configurations of the automaton. In order to see how these matrices affect single qubits, we rewrite (4.3), using (2.7), as

$$F_{XY} = \bigotimes_{\substack{\text{cell (x,y) in} \\ \text{block (X,Y)}}} m_{xy}, \quad \text{where } m_{xy} = \begin{cases} R, & \text{if block } (X,Y) \text{ is an even block} \\ L, & \text{if block } (X,Y) \text{ is an odd block} \\ 2\times 2 \text{ identity matrix}, & \text{otherwise.} \end{cases} \tag{5.1}$$

The matrices A_{XY} are left out, since we are only studying the guard bits. Each m_{xy} transforms one qubit of the configuration. Now we see how F_{XY} transforms a basis state either into some other basis state or into a null vector. The result is a basis state, if the initial state had four $|1\rangle$'s ($|0\rangle$'s) at certain positions, corresponding to cells of an even (odd) block of the automaton. If even one of those necessary qubits is in a wrong state, the raising or lowering operator gives a null vector, taking the whole tensor product to null.

The resulting Hamiltonian can be interpreted as a relation, which tells whether a basis state is reachable from another or not. This is because it consists of ones and zeros only, as can be seen by the following reasoning. The full Hamiltonian (4.2) is formed by summing terms F_{XY} and their conjugates. Each summand has only 1's and 0´s as its elements (because the columns are either null or automaton basis states). Also the result will have only 1's and 0's in it. Different summands can determine different behaviors for the same basis state, in situation where the state happens to have several blocks with four 1's or four 0's. However, taking a basis state and flipping some bits on different positions, necessarily leads into different basis states, and thus no two summands will have a 1 in exactly the same column and the same row.

It is important to notice that H specifies some reaction for all possible guard qubit configurations, which can be formed from $|1\rangle$ and $|0\rangle$ states on the automaton grid. This means that the matrix size for an $rows \times cols$ grid is $2^{rows*cols} \times 2^{rows*cols}$ — for example, the behavior of 4×4 guard bit grid states would be fully described by a matrix with 65536 rows and columns.

However, when operating the automaton (classical as well as quantum) we don't necessarily use all possible configurations. We have an initial state, which can develop into some other, but not necessarily to all possible, states. Of course, the cell contents of the data blocks can have different initial values for different runs, but the initial state (and thus the set of reachable states) of the guard bit part will always be the same. Thus describing the behavior for never-encountered configurations is unnecessary, and the columns specifying their successors could be left out of the Hamiltonian matrix without losing any necessary information.

Now the question is, how can we find only necessary configurations among, say, 65536 possible? Writing out and analyzing huge matrices is clearly not the most practical solution. What we can do instead is to model the behavior of the locally synchronized, reversible classical automaton as a Petri net. The reachability graph of the net model will contain just the reachability information we want to give the quantum automaton as its Hamiltonian.

The net model can be built as follows. The cells of the automaton are represented by places, and the initial state of the automaton becomes the initial marking of the net. For each term F_{XY} the net has a transition, which has arcs (both input and output) to four places. The transition can fire only, if the input places are all $\langle 0\rangle$ (for transitions of odd blocks) or all $\langle 1\rangle$ (for even block transitions). The transition flips the qubits by outputting $\langle 1\rangle$ (odd blocks) or $\langle 0\rangle$ (even blocks) into the places it handles. The reverse behavior, given by $F_{XY}^{\dagger}$, is similarly implemented as transitions. The reverse transitions behave like the F_{XY} transitions, but with roles of $\langle 1\rangle$ and $\langle 0\rangle$ reversed.

The net can be folded by making the cell coordinates parameters. After folding, the net will have one place, for holding the cell values, and four transitions, for representing the forward and backward behavior, separately for even and odd blocks. The net model for the classical MCA is given in Figure 5.

The only place of the folded net represents the guard bit configurations of the automaton. The logical steps between the configurations can be read directly from the reachability graph.

The rows of the full MQCA Hamiltonian were numbered from $|11\ldots11\rangle$ to $|00\ldots00\rangle$. To preserve the original ordering of the rows and columns of the Hamiltonian, we label also the reachable configurations with binary numbers in descending order.

The only difference if that the matrix we are building will contain a row for each reachable configuration only, and thus not all binary numbers are used. The row numbers (starting from 1) of this smaller matrix will be used as guard state numbers in the final model.

We can build our Hamiltonian from the reachability graph by making element at row r in column c a "1" if the graph says it's possible to move from configuration c to configuration r. The reachability graph for the example automaton and the corresponding Hamiltonian are shown in Figure 6.

It should be kept in mind that although we use a smaller matrix, and refer to it as

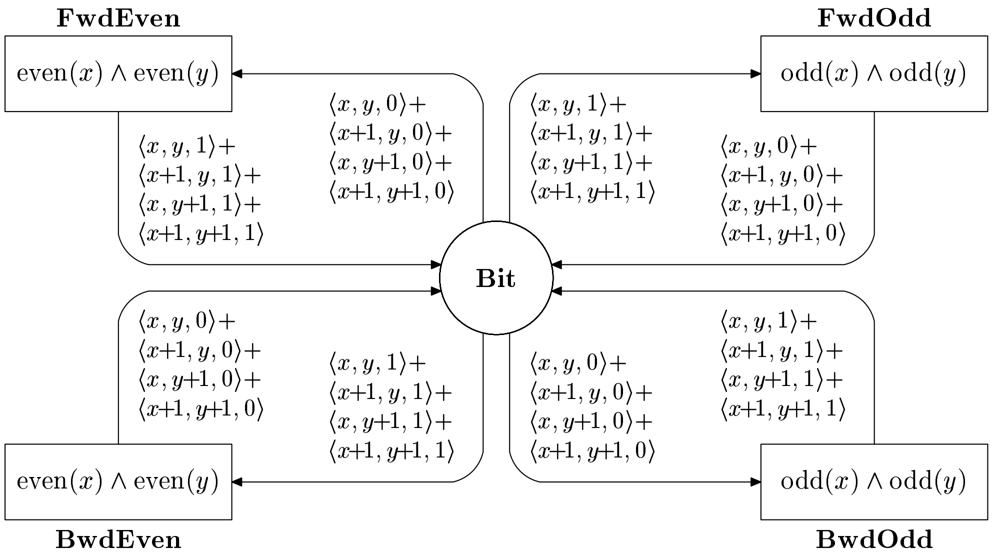


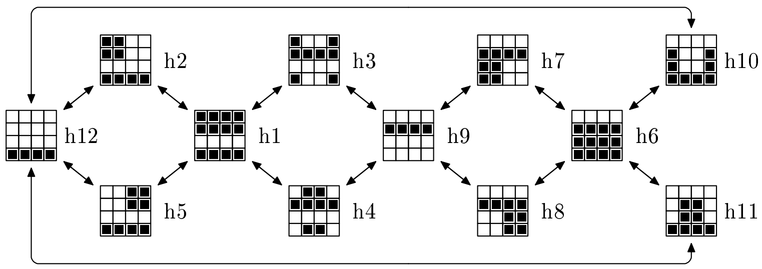
Figure 5: The guard bit system of MCA represented as a Petri net. If the MCA cell located at (x_i, y_i) has initially value v_i (which must be either 0 or 1), then the initial marking of the net will have token $\langle x_i, y_i, v_i \rangle$.

"the Hamiltonian" of the quantum automaton, the minimization procedure we use is only a trick which helps us to analyze the behavior we are interested in. This minimization procedure does not anyhow affect the possible physical realization of the automaton.

5.2 Reachable states in MCA and MQCA

The automaton we chose as our example is quite big for a quantum system. Generally, computing the behavior of a 32 qubit system would require generating and multiplying $2^{32} \times 2^{32}$ matrices. However, when the computation of the automaton is guided and restricted by a guard qubit system, the number of the basis states actually used drops dramatically – in our example into 36 states.

If the initial states of all the guard bits were set to 0, the quantum automaton would carry out essentially the same computation as the classical one, in the sense that the possible moves between data configurations would be the same. The computational power of the model is not affected by the choice of the initial guard bit states (in [13] Margolus points out that an automaton with his special initial state could be used e.g. for building digital circuits). However, as long as we use the special initial state for the guard bits, we cannot directly simulate a classical automaton with a quantum automaton of the same size, because the special initial state reduces the state space of the automaton (see Figure 7). The difference is illustrated by the example computation chosen. The classical version simulates a collision of two particles: they start from corners, approach each other and collide, continuing into different directions. In the version with Margolus's initial state, the particles move but the collision never takes place.



$$
\begin{bmatrix}
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0
\end{bmatrix}
$$

Figure 6: Reachability graph of the guard bit system and the corresponding Hamiltonian. Numbering of the states in the graph corresponds to the order of rows and columns in the matrix.

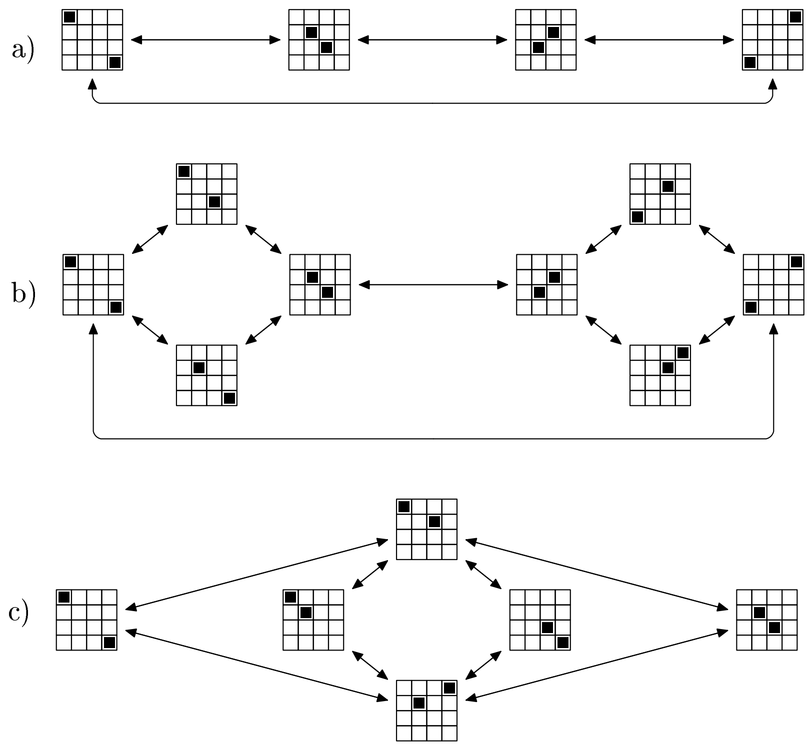


Figure 7: Behavior of data configurations for a) globally synchronized MCA, b) locally synchronized MCA with guard bits initially set to zero, and c) locally synchronized MCA with initial state as in Figure 2. Only the data part is shown, but the behavior is guided by the guard bit part as usual.

6 Modeling the operation of MQCA

6.1 How the probability amplitudes are computed

Above, we studied the behavior of the guard qubit system. We computed the reachability graph for guard qubit states, and represented it in the form of Hamiltonian matrix H. Now it's time to take a look at how a quantum computer performs a computation defined by this Hamiltonian.

During a computation step of length Δt, an initial state $|\psi(0)\rangle$ of the computer is transformed into a new state $U_{step}|\psi(0)\rangle$. The transformation is done by an evolution operator (given in (2.13))

$$U_{step} = \sum_{k=0}^{\infty} c_k H^k, \qquad c_k = \frac{(-\mathrm{i}\Delta t)^k}{k!}. \tag{6.1}$$

As we saw in Section 5.1, the Hamiltonian H of the MQCA corresponds to the reachability graph, which tells how the basis states of the computer are connected to each other. The Hamiltonian has value 1 at entry (row, col), if there is an edge between basis states row and col in the graph. Mathematically, moving from an initial state into a superposition of all neighboring states is represented by multiplying the initial state vector by H. If we multiply the initial state by H^k, the resulting state vector is a superposition of all basis states reachable from the initial state by taking k steps in the reachability graph. In (6.1) we see that all possible paths, up to those of infinite length, are considered when forming the evolution operator.

Because of the reversibility of quantum computations, there are no states with zero successor states. If we can arrive into a state, we can also get out, at least by returning the same way we came. When there is always a way to take one more step, even the highest powers of H don't vanish. Therefore, when computing U_{step}, we must handle an infinite number of terms in (6.1).

Generally, the new state $U_{step}|\psi(0)\rangle$ is a superposition of several basis states. Each path ending in some basis state affects its probability amplitude. The longer the path, the lesser its effect, as we can see by looking at coefficients c_k: when k grows, the absolute value of c_k decreases. As the coefficients are complex numbers, the different paths can cancel or reinforce each other, a phenomenon known as quantum interference.

If the guard bit system was our sole object of interest, we could simply compute the evolution of the system using (6.1), without building any further models. We are, however, more interested in bringing forth the interplay between data qubits and guard qubits. Therefore, we want to build our net model so that it shows the details of computation done in (6.1). That way, we can see how changes in data qubits follow the changes in guard qubits. We would also like our model to reflect the fact that even when the actual computation (determined by initial values of data qubits) changes, the guard bit part remains the same. This is another reason why we handle guard qubits and data qubits separately.

To accurately describe the operation of the computer, we would need to handle an infinite number of paths. As we saw in (6.1), the longer paths have less effect on the result than the shorter ones. Therefore, we should be able to get good enough results by considering only paths shorter than some (arbitrarily chosen) limit $LIMIT$:

$$U_{appr} = \sum_{k=0}^{LIMIT-1} c_k H^k. \tag{6.2}$$

When generating the paths, we can use a previously generated shorter path when building the longer ones. (It should be kept in mind, however, that although a longer path contains a shorter one, both paths have separate effect in the computation, with different coefficients c_k.) This corresponds to doing the computation of $U_{appr}|\psi(0)\rangle$ in order

$$[(((\cdots + I)\frac{-\mathrm{i}\Delta t}{3}H + I)\frac{-\mathrm{i}\Delta t}{2}H + I)\frac{-\mathrm{i}\Delta t}{1}H + I]|\psi(0)\rangle. \tag{6.3}$$

In (6.1) and (6.2) the paths of the same length were grouped together in the same term. In (6.3), multiplication by the identity matrix I corresponds to ending a path in some guard bit state, and H terms generate longer paths by taking a step into the neighboring states. Also the coefficients c_k can be formed recursively as

$$c_k = \frac{-\mathrm{i}\Delta t}{k} c_{k-1}, \qquad c_0 = 1. \tag{6.4}$$

The H matrix does not contain any information of the data qubits. However, the operation of the automaton is defined so that a block of data qubits can change only, when the corresponding block of guard qubits is updated. Because of this, the necessary data qubit states are generated along with changes of guard qubits. The probability amplitudes are now computed for a basis state, which is formed both from guard and data qubit values.

The basic idea for implementing all this as a net model is as follows.

Each path we are following is represented as a token, which we call a *path token.* A path token contains information about the *guard state* (the state of the guard qubits), the *data configuration* (the state of the data qubits), the current coefficient c_k (termed *probability amplitude of the path*, because the probability amplitudes are formed by summing together these coefficients) and the path length.

Steps forward on a path are taken by transitions. First, the old path token is saved as a *ready token.* A ready token represents a path which ended in the current state. Second, the path length is checked to see if it has reached the limit $LIMIT$. If not, longer paths are generated. This is done by creating new path tokens, one for each state reachable by one step from the current state. The coefficients of the new paths are smaller than that of the old path, and are computed using (6.4).

Finally, the ready tokens are checked to see, whether two paths ended in the same state (that is, whether they had both the same guard state and the same data configuration). Two such tokens are replaced by one token, with the amplitudes of the two paths summed. When no more ready tokens can be combined this way, every basis state which can be part of the current quantum state is represented by one ready token. Together, the set of ready tokens represents the superposed state.

6.2 Overview of the net model

Figure 8 is a top-level view of the net model. Most of its transitions represent subsystems consisting internally of several places and transitions. The arc expressions tell which effect

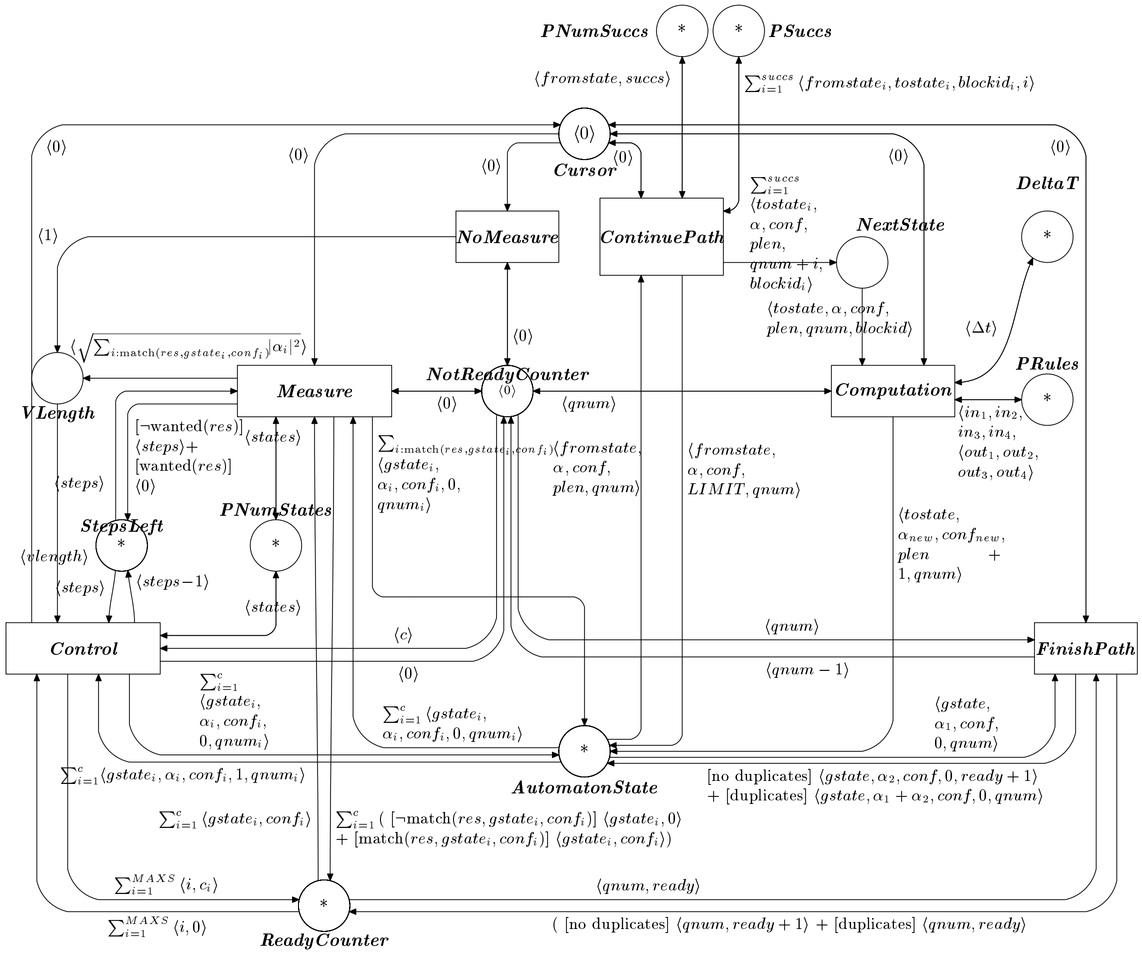


Figure 8: Overview of the model. For the initial markings of places marked with *, see Figure 9.

the whole subsystem has on the places shown in the picture, and therefore some of them contain expressions which could not be implemented using one Pr/T transition only.

Subsystems ***ContinuePath*** and ***FinishPath*** take care computing the paths (branching to longer paths and saving the current path token). ***Computation*** updates the data configurations according to guard bit changes. ***Control*** takes care of finishing a time step and preparing the net for a new one. ***Measure*** models the measurements. ***NoMeasure*** is not a subsystem, but a normal transition. It fires in conflict with the *Measure*. It prepares the net for the operation of *Control*, like *Measure* does, but does not do the measurements.

The heart of the model consists of three places: ***AutomatonState***, ***ReadyCounter*** and ***NotReadyCounter***. Place *AutomatonState* holds the path tokens and ready tokens (although they travel through several other places, too, during the computation). The path tokens are of form $\langle gstate, \alpha, conf, plen, qnum\rangle$. Ready tokens are similar to path tokens, but their path length *plen* is zero. Since we know all possible guard states already, they can be represented in a path token simply as a state number *gstate*. About possible data configurations we don't know anything, and thus the data bit configuration is completely

$AutomatonState$	$\langle 1,12,1.0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0\rangle$
$ReadyCounter$	$\langle 12,1\rangle + \langle 1..11,0\rangle$
StepsLeft	number of steps to do, e.g. $\langle 5\rangle$
$DeltaT$	time step length, e.g .$\langle 0.25\rangle$
PRules	$\langle 0,0,0,0,0,0,0,0\rangle+\ \langle 1,0,0,0,0,0,0,1\rangle+$ $\langle 0,1,0,0,0,0,1,0\rangle+\ \langle 0,0,1,0,0,1,0,0\rangle+$ $\langle 0,0,0,1,1,0,0,0\rangle+\ \langle 1,0,0,1,0,1,1,0\rangle+$ $\langle 0,1,1,0,1,0,0,1\rangle+\ \langle 1,1,0,0,1,1,0,0\rangle+$ $\langle 0,0,1,1,0,0,1,1\rangle+\ \langle 1,0,1,0,1,0,1,0\rangle+$ $\langle 0,1,0,1,0,1,0,1\rangle+\ \langle 0,1,1,1,0,1,1,1\rangle+$ $\langle 1,0,1,1,1,0,1,1\rangle+\ \langle 1,1,0,1,1,1,0,1\rangle+$ $\langle 1,1,1,0,1,1,1,0\rangle+\ \langle 1,1,1,1,1,1,1,1\rangle)$
PNumStates	$\langle 12\rangle$
PNumSuccs	$\langle 1,4\rangle\ +\ \langle 2,2\rangle\ +\ \langle 3,2\rangle\ +\ \langle 4,2\rangle\ +$ $\langle 5,2\rangle\ +\ \langle 6,4\rangle\ +\ \langle 7,2\rangle\ +\ \langle 8,2\rangle\ +$ $\langle 9,4\rangle\ +\ \langle 10,2\rangle+\ \langle 11,2\rangle+\ \langle 12,4\rangle$
PSuccs	$\langle 12,2,11,1\rangle\ +\ \langle 12,5,13,2\rangle\ +\ \langle 12,10,24,3\rangle\ +$ $\langle 12,11,22,4\rangle\ +\ \langle 2,12,11,1\rangle\ +\ \langle 2,1,13,2\rangle\ +$ $\langle 5,12,13,1\rangle\ +\ \langle 5,1,11,2\rangle\ +\ \langle 1,2,13,1\rangle\ +$ $\langle 1,5,11,2\rangle\ +\ \langle 1,3,42,3\rangle\ +\ \langle 1,4,44,4\rangle\ +$ $\langle 3,1,42,1\rangle\ +\ \langle 3,9,44,2\rangle\ +\ \langle 4,1,44,1\rangle\ +$ $\langle 4,9,42,2\rangle\ +\ \langle 9,3,44,1\rangle\ +\ \langle 9,4,42,2\rangle\ +$ $\langle 9,7,31,3\rangle\ +\ \langle 9,8,33,4\rangle\ +\ \langle 7,9,31,1\rangle\ +$ $\langle 7,6,33,2\rangle\ +\ \langle 8,9,33,1\rangle\ +\ \langle 8,6,31,2\rangle\ +$ $\langle 6,7,33,1\rangle\ +\ \langle 6,8,31,2\rangle\ +\ \langle 6,10,22,3\rangle\ +$ $\langle 6,11,24,4\rangle\ +\ \langle 10,6,22,1\rangle\ +\ \langle 10,12,24,2\rangle\ +$ $\langle 11,6,24,1\rangle\ +\ \langle 11,12,22,2\rangle$

Figure 9: Initial markings for parameter places and other places which determine, what kind of computation is done on the automaton.

given by listing the values of all data cells. We write this list out explicitly only when we need to access the values of individual data qubits; usually, we use one variable name (here $conf$) to refer to the whole list. The amplitudes α are complex numbers.

Tokens $\langle gstate, ready\rangle$in place *ReadyCounter* hold the number of ready tokens, separately for each guard state. The number of unfinished paths $\langle notr\rangle$, is kept in place *NotReadyCounter*. This count includes all the path tokens currently found in the net, not only those in $AutomatonState$. Every time a subsystem creates, modifies or combines path tokens, it updates the values in *ReadyCounter* and *NotReadyCounter* correspondingly.

The initial markings of these places determine the initial state of the automaton. Usually, the initial state is a basis state (not a superposition), and $AutomatonState$ has only one ready token, $\langle startstate, 1, startconf, 0, 1\rangle$. Correspondingly, *ReadyCounter* contains tokens $\langle state, 0\rangle$ for all other states, and $\langle startstate, 0\rangle$ for the initial state. Place *NotReadyCounter* is initially $\langle 0\rangle$.

The amplitudes computed in the net are formed by summing the amplitudes which we got by following different paths. As addition is a commutative operation, it does not matter in which order we follow and sum the different paths. Also the relative order of starting a new path or continuing the computation on a previous one is unimportant. Without special restrictions, however, any analysis tool handling the net model would study all possible orders of events – effectively performing the same computation several times. This is unnecessary, because we know that studying one (arbitrarily chosen) case would be enough to give us the results. Furthermore, as the paths branch from each state into all successor states, the reachability graph with several copies of the same computation would grow so rapidly that it would practically prevent us from getting any results out of the net.

In order to get rid of computation of the unnecessary cases, we fix an order for handling the path tokens. Each time a path token is created, it gets a *queue number*, which is the value in *NotReadyCounter* increased by one. A subsystem can handle only tokens whose queue number matches the counter in *NotReadyCounter*. That is, the tokens are handled in last-come-first-served order.

Although the relative operating order of different subsystems is not important, two subsystems handling the same counter places at the same time might disturb each other. Therefore each system must seize a token $\langle 0 \rangle$ from place ***Cursor*** before its transitions can fire. When the subsystem is ready, it returns $\langle 0 \rangle$ to *Cursor* for other subsystems to use. A $\langle 0 \rangle$ token is used instead of a plain token, because *ContinuePath* subsystem needs to use also nonzero values in the cursor place.

The structure of the guard bit system and the block rules used in computation can be set by choosing a suitable initial marking for a number of parameter places. The role of these places is only to provide transitions with necessary information about the computer and computation task, and their markings never change. The initial markings of the parameter places for the example automaton are shown in Figure 9.

The length of the time step is put into ***DeltaT***. The number of possible guard bit states is stored in ***PNumStates***. Tokens in ***PNumSuccs*** tell for each guard bit state how many successor state it has. Place ***PSuccs*** lists these successor states. In effect, *PSuccs* and *PNumSuccs* encode the Hamiltonian for the guard bit system.

Place ***PRules*** contains the block rules in form $\langle in_1, in_2, in_3, in_4, out_1, out_2, out_3, out_4 \rangle$. They give the initial and final state of a 2×2 block, with cell values listed from left to right and top to bottom.

Initial marking of place ***StepsLeft*** tells the model, how many computation steps must be done. The marking of the place always equals the number of computation steps left, and is updated by *Control* subsystem.

The place ***NextState*** acts as intermediate place for passing path tokens between two subsystems. Subsystem *ContinuePath* puts there path tokens. Compared with the path token stored in *AutomatonState*, these tokens have an additional field telling which guard bit block was updated to get to the guard bit state mentioned in the token. Subsystem *Computation* uses this information to make corresponding update to the data bit configuration.

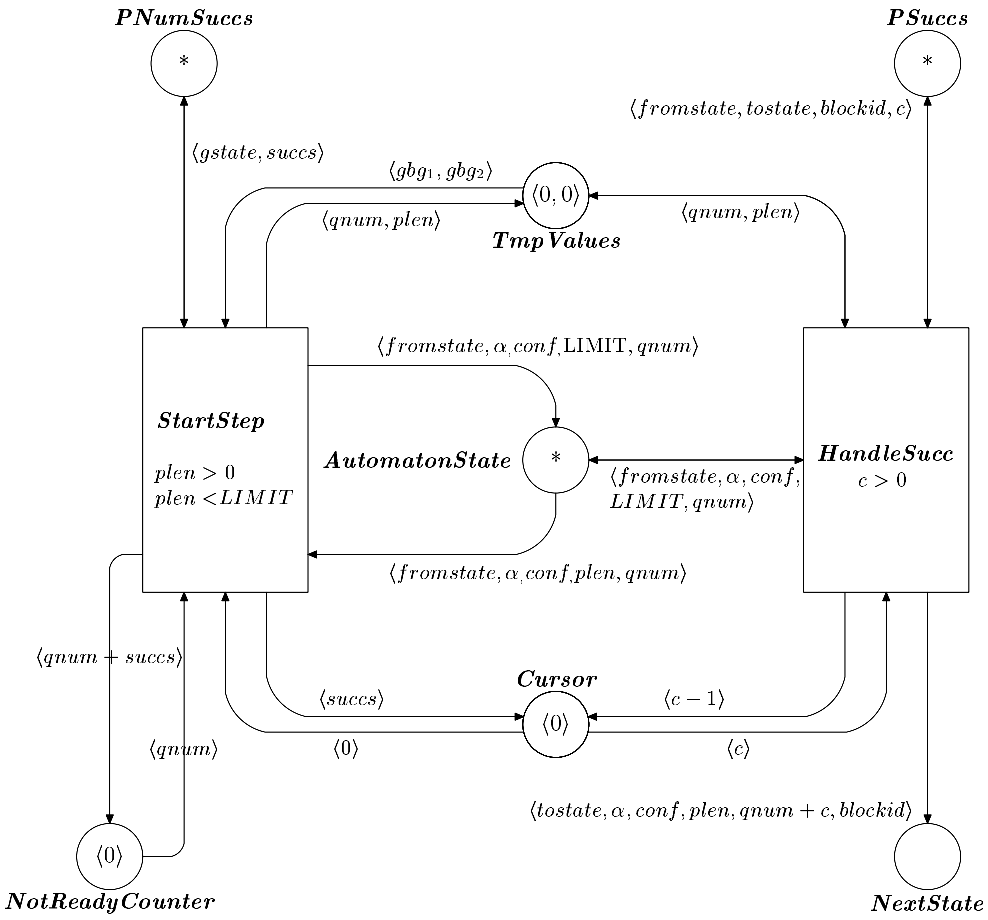


Figure 10: *ContinuePath* subsystem. For the initial markings of places marked with *, see Figure 9.

6.3 Taking a step on a path

Briefly, the operation of *ContinuePath* subsystem consists of taking a path token from *AutomatonState*, finding out into which guard states the next computation step can be taken, marking the current path token as finished and producing new path tokens for successor states. This subsystem is shown in Figure 10.

Because different guard states can have different numbers of successors, it is not enough simply to use one transition which dumps new tokens into the net. Therefore, *ContinuePath* consists of two transitions: ***StartStep*** picks from *AutomatonState* a path token (the one which corresponds to the current queue number in *NotReadyCounter*) and reads the number of the successor states from *PNumSuccs*. It then initializes a loop, by storing the number of successors into *Cursor* and the current queue number and path length into an auxiliary place *TmpValues*. Another transition ***HandleSucc*** loops through the successors and creates a path token for each, putting them into the place *NextState*.

HandleSucc can fire only as long as *Cursor* has values greater than zero. Each time it fires it decrements *Cursor* by one. Using the cursor place this way as a counter stops the looping automatically when all the values have been handled. When the value in *Cursor* reaches zero, *HandleSucc* is no more enabled, but other subsystems can operate because they see the $\langle 0 \rangle$ token in *Cursor*.

Initially, the auxiliary *TmpValues* contains a token of two entries, but this token is needed only to enable *StartStep*, which then puts meaningful values into *TmpValues*. Likewise, after the loop has finished, some garbage remains in *TmpValues*, and is removed by *StartStep* next time it fires.

The description of the automaton is read from *PNumSuccs* and *PSuccs*. Tokens in *PNumSuccs*, of form $\langle gstate, succs \rangle$, simply list the guard states and the number of successors for each. The tokens $\langle fromstate, tostate, blockid, succnum \rangle$ in place *PSuccs* describe the logical steps between any two guard states by listing the initial and the final states, the number of the automaton block which is updated when moving between the states, and the number of successor. The successor number is used to identify different successors of the same state, so that each successor is used only once in the loop. The information about block updated is not used here, but is stored in tokens $\langle gstate, \alpha, conf, plen, qnum, blockid \rangle$, which are put into *NextState* for the *Computation* subsystem to use.

In addition of initializing the loop, which creates new longer paths, *StartStep* also saves the current situation by replacing the path token it took from *AutomatonState* by a similar token, but with *plen* set to $LIMIT$. This prevents *StartStep* from handling it further (the transition can fire only if the path length of the token is above zero and below $LIMIT$) and later also tells the *FinishPath* subsystem that it may handle the token. The place *NotReadyCounter*, which must always hold the number of tokens which still need some attention, is increased by *succs*.

6.4 Finishing a path

Finishing the computation of a path is conceptually simple – just setting the path length to zero to prevent further handling, and combining the amplitudes with those paths generated earlier, if several paths happen to end in the same guard state with the same data configuration. Implementing this behavior in the net model, however, requires several places and transitions. The *FinishPath* subsystem is shown in Figure 11. The key problem in summing the amplitudes is that in order to add the amplitude of a newly arrived token into the older amplitude, we must first find out whether that older exists. As we have no a priori knowledge of the states into which the data system can develop, we have also no way of knowing whether there exist any older tokens with the same guard bit and data bit values or not. Our net formalism does not permit us to simply take a look at *AutomatonState* to check for the existence of a token. Instead, we must do this checking by using additional places, which keep record of tokens in *AutomatonState*.

The straightforward solution would be to use the knowledge of the number of tokens in *AutomatonState*, and every time a new token is ready, to compare it to all old tokens. Since this would cause quite a lot of unnecessary comparisons, we use a slightly more complicated scheme in our model. Instead of comparing a new token to all ready tokens we compare it only to those tokens with the same guard state. This requires keeping separate token counters for each of the guard states, which we can quite easily do, since we know all the possible guard bit states. These counters are stored into place *ReadyCounter*. The place always holds a token for each guard state.

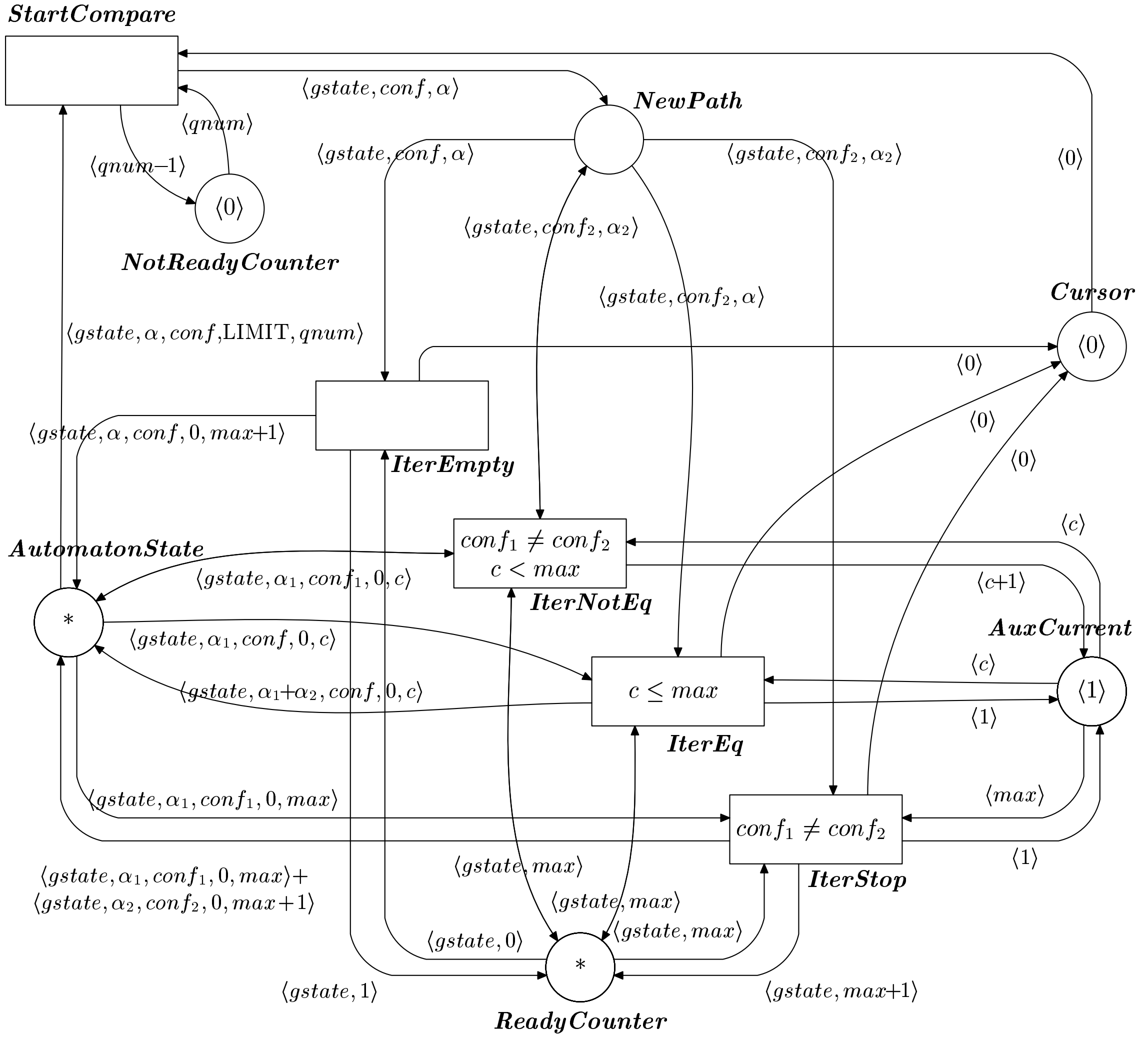


Figure 11: *FinishPath* subsystem. For the initial markings of places marked with *, see Figure 9.

The operation of *FinishPath* starts with firing of ***StartCompare***. It removes the $\langle 0 \rangle$ token from *Cursor* to prevent other subsystems from operating while *FinishPath* is doing its comparisons. It also moves the current path token (the one with the queue number matching the value in *NotReadyCounter*), from *AutomatonState* to ***NewPath***, decreases the number of unfinished paths in *NotReadyCounter*.

When the input token $\langle gstate, \alpha, conf \rangle$ has appeared in *NewPath*, transitions ***IterEmpty***, ***IterNotEq***, ***IterEq*** and ***IterStop*** compare its contents to all tokens in *AutomatonState* with the same state number. Transition *IterEmpty* handles the situation with no old tokens to compare to, i.e. *ReadyCounter* contains a token $\langle state, 0 \rangle$ for the current state. In this case, a new path token is put to *AutomatonState*, with zero path length and queue number set to 1, the ready token counter in *ReadyCounter* is updated, and a $\langle 0 \rangle$ token is returned to *Cursor* to tell that *FinishPath* is ready.

Transitions *IterNotEq* and *IterEq* do the comparisons. *IterNotEq* fires, if the old path

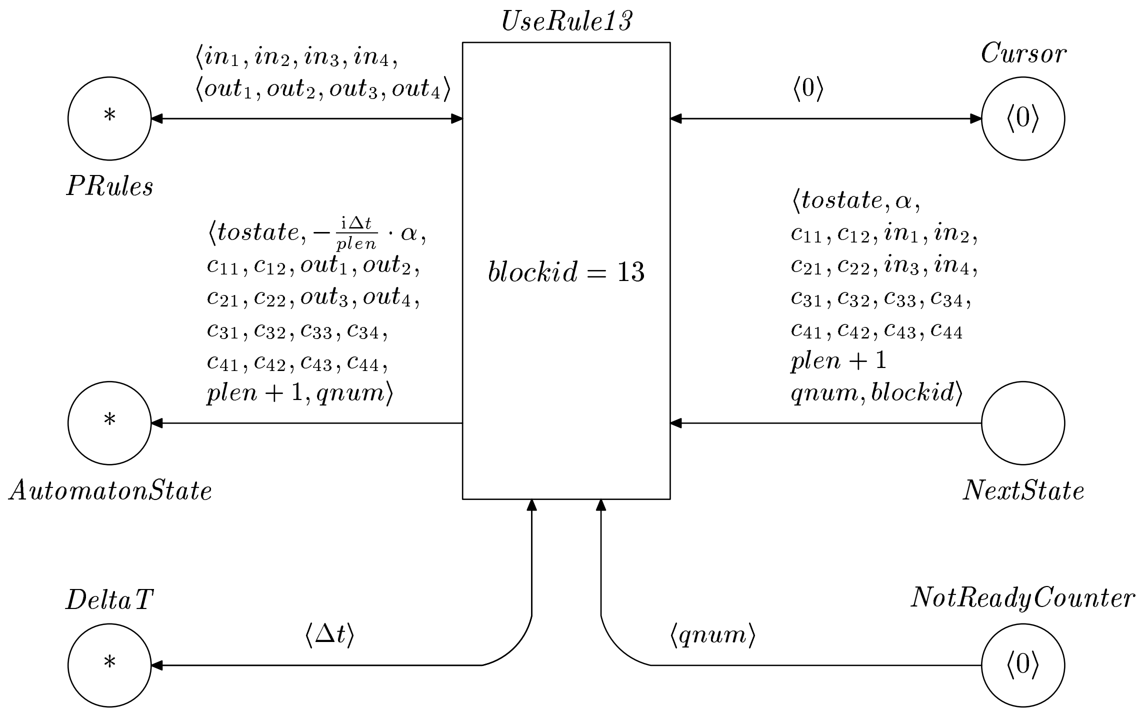


Figure 12: One transition of *Computation* subsystem. There is a similar transition for each block of the automaton. For the initial markings of places marked with *, see Figure 9.

token, with queue number matching that of ***AuxCurrent***, is found to differ from the new path token. In this case, only the counter in *AuxCurrent* is increased, values in other places don't change. If the old and the new token have the same data configuration, *IterEq* fires. It replaces the old path token in *AutomatonState* with a new one, with an amplitude $\alpha = \alpha_{old}+\alpha_{new}$. It also resets the values in *AuxCurrent* to 1, for the next comparison round to use, and returns the $\langle 0\rangle$ token to *Cursor*.

Transition *IterStop* stops the iteration in the case where there are some old path tokens with the same guard state, but none of them matches the data configuration of the new token. The new path token is moved to *AutomatonState*. The ready token counter of the guard state in *ReadyCounter* is increased, and its new value also serves as the queue number of the newly created path token. Finally, the $\langle 0\rangle$ token is returned to *Cursor*.

6.5 Computing with data bits

The behavior of classical automaton was defined so that a change in the data bits is always followed by flipping of the corresponding guard bits. This same behavior is present in MQCA: in (4.3), the rule matrices A_{XY} have an effect only, if the guard qubit operators produce a non-null vector. Because of this, we don't need to build the complete Hamiltonian for the computer in order to compute the data configurations. The data can change only if the guard bits change, and thus it is enough to model both changes together.

Changes in amplitudes and data configurations are handled in *Computation* subsystem. This is the only part of the model, which must be rewritten if the size of the automaton changes; other changes, e.g. the Hamiltonian of the guard bit system, can be changed by changing the initial markings in parameter places. *Computation* contains a transition ***UseRuleXY*** for each block (X, Y) of the automaton. All transitions behave in similar fashion, but they update different parts of the data bit configuration. An example transition, which handles the block (1,3) of the the example automaton, is shown in Figure 12.

The data configuration entries in path tokens (which earlier were shown as a single $conf$ entry) are now written out to show the data configuration in detail. Transition *UseRuleXY* takes from *NextState* a path token with some data configuration. It then reads from *PRules* the block rule, whose input bits match those bits of the configuration, which reside in the block (X, Y). To *AutomatonState* it outputs a token with configuration, where the critical bits have been replaced by the output values of the rule.

In addition, *UseRuleXY* multiplies the amplitude by coefficient

$$c_{pathlength} = \frac{-\mathrm{i}\Delta t}{pathlength}, \tag{6.5}$$

to implement the decreasing effect of longer paths, as the coefficients c_k do in (6.3).

6.6 Controlling the computation

The computation of a time step is finished, when there are no more unfinished path tokens in the net (*NotReadyCounter* contains a token with value 0). Then the *Control* subsystem (in Figure 13) prepares the net for the next time step, if one is needed.

When *Control* starts working, either *NoMeasure* or some transitions of *Measure* have just fired, putting a token into ***VLength***. Transitions ***NextToken*** and ***CNextState*** iterate through the tokens in *AutomatonState*, marking their path lengths to 1 and updating the counters in *ReadyCounter* and *NotReadyCounter*. They use *CIterState* as a current state counter and the place ***CIterMoved*** for counting how many data configurations of the current guard state have been handled. The amplitudes are divided by the value in *VLength*, which corresponds to normalizing the state vector after measurement. If the state was not measured, *NextStep* has set *VLength* to $\langle 1 \rangle$ so that this normalization has no effect.

Transition ***NextStep*** compares the token $\langle states \rangle$ in *PNumStates*to the current state counter in *CIterState*. When all the path tokens have been handled, the counter has reached the value $\langle maxstatenum + 1 \rangle$. Now all the values in the net are set ready for the next time step, and, provided there are steps left to perform in the *StepsLeft*, *NextStep* starts a new step by returning the cursor token to *Cursor*.

6.7 Measurements

The time needed for making a measurement is assumed to be negligibly small, so the measurements in the model can safely take place between the time steps, without consuming any time themselves. The transition *NoMeasure* (in the top-level view in Figure 8) represents a situation when a measurement is not made, and performing a measurement is modeled in *Measure* subsystem (Figure 14).

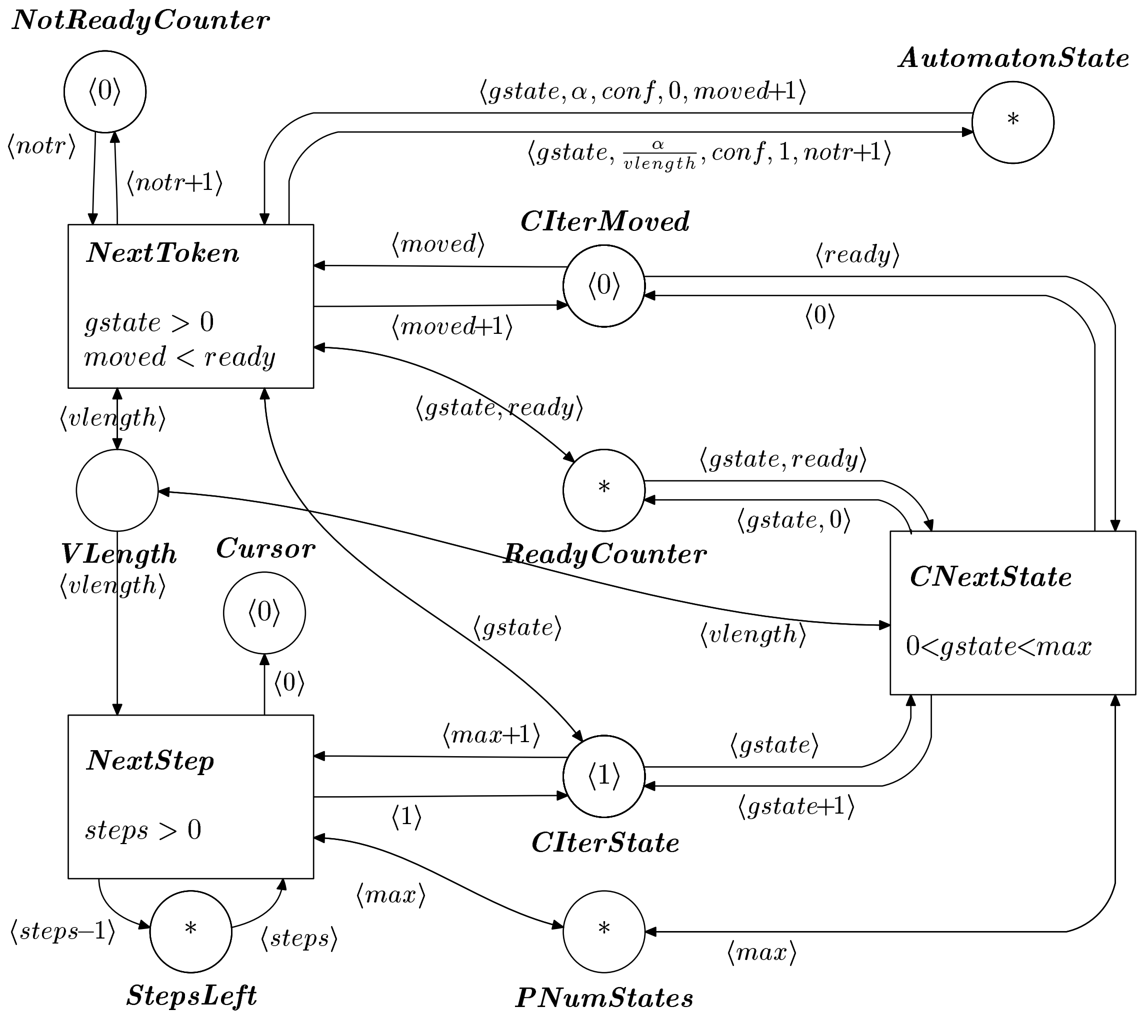


Figure 13: *Control* subsystem. For the initial markings of places marked with *, see Figure 9.

We assume that the length of the computational step is chosen to be the shortest desired time between measurements. After all, there is no reason to divide the computation into smaller steps, especially in this type of model where inaccuracies of computation accumulate when more steps are taken. Thus we don't use any firing conditions in the measurement transition, but let it fire after each step (in conflict with transition *NoMeasure* which represents the situation when the computation continues without a measurement).

As we are building our model rather for analyzing than for simulation, the amplitudes, which in real life would determine the probability of measurement outcomes, don't affect the behavior of the transitions in the model. If a basis state is present in the superposed state, even with a small amplitude, it can be part of the automaton state after a measurement. In the model, the ready tokens correspond to the basis states, and all the ready tokens in *AutomatonState* form the superposed state. The measurement result is selected by reading one path token from *AutomatonState*, and storing its measured qubits into ***Result***.

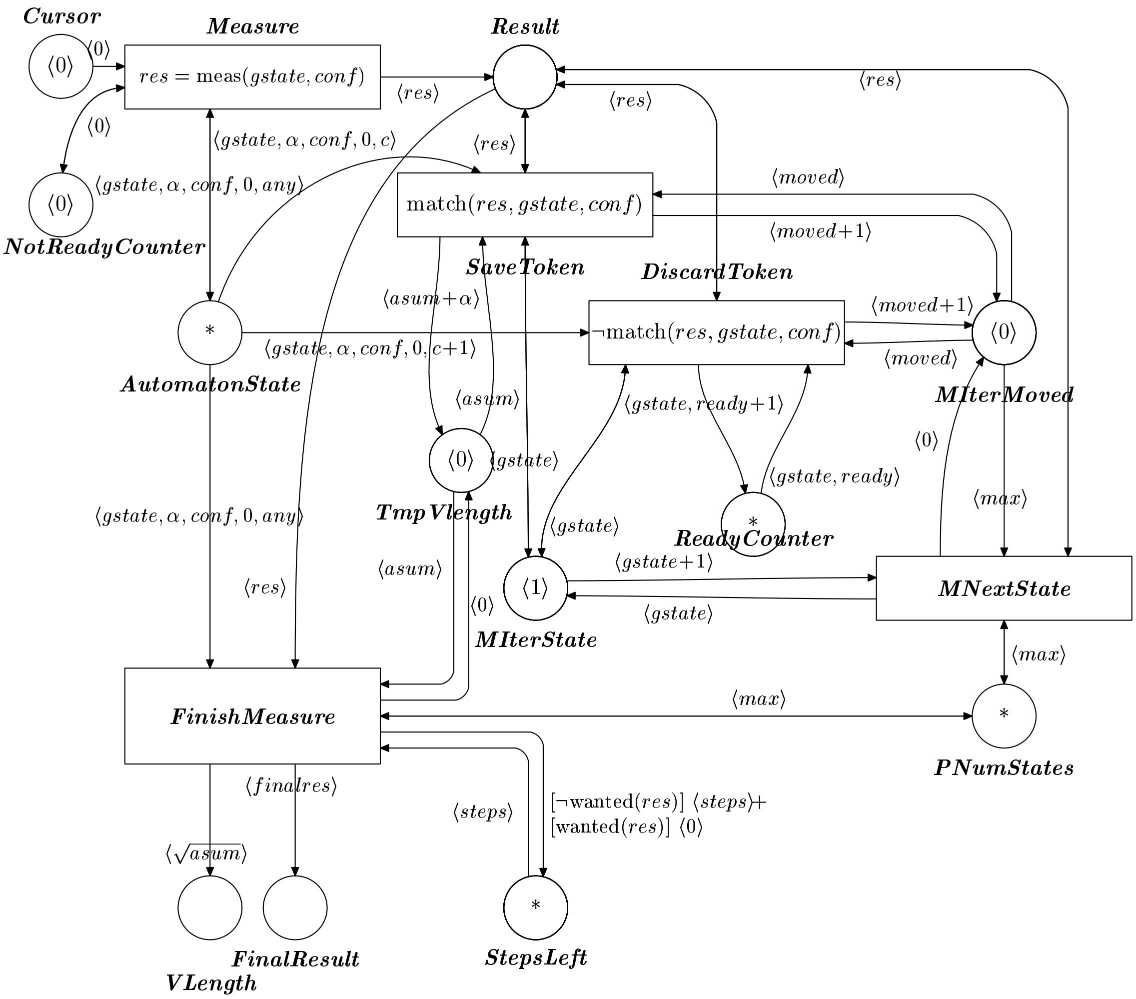


Figure 14: *Measure* subsystem. For the initial markings of places marked with *, see Figure 9.

Exactly which qubits of the state are measured is determined by two functions. The arguments of $\mathrm{meas}(gstate, c_1, c_2, \ldots, c_n)$ represent a basis state of the automaton (both the guard state and the data configuration), and it returns the measured parts of the state. Function $\mathrm{match}(res, gstate, c_1, c_2, \ldots, c_n)$. compares the corresponding parts of a basis state to the measurement result, and returns true if they are the same.

For example, in the example automaton we have decided to measure the sixth qubit of the data configuration, thus we define the functions as

$$\mathrm{meas}(gstate, c_1, c_2, \ldots, c_n) = c_6$$

and

$$\mathrm{match}(res, gstate, c_1, c_2, \ldots, c_n) = \begin{cases} \text{true}, & \text{if } c_6 = 1 \\ \text{false}, & \text{otherwise} \end{cases} .$$

When the state is measured (in the model, ***Measure*** chooses the measurement result) the state collapses. In the net model, the collapse happens by removing from *AutomatonState* all tokens, with are incompatible with the measurements results (those which have some qubits with values different from those measured).

This is done by looping through all the tokens. ***SaveToken*** fires for those tokens, which can be part of the measured state (for them, match(...) returns true), and ***DiscardToken*** for the others. Transition *DiscardToken* just eats tokens and decrements the counter in *ReadyCounter*. *SaveToken* does not change contents of *AutomatonState* or the token counter. Instead, it updates *TmpVlength* by adding to its old value (which was set to $\langle 0 \rangle$ by *Measure* before the looping started) the squared magnitude of the amplitude. The loop formed by *SaveToken*, *DiscardToken* and ***MNextState*** works similarly to the loop in *Control*. It uses ***MIterState*** and ***MIterMoved*** as its counter places, *Measure* initializes it and *FinishMeasure* notices when the loop is over.

After the collapsing phase has checked all the states, ***FinishMeasure*** fires. It completes the computation of the state vector length by taking token $\langle vlength \rangle$ from *TmpVlength* and putting $\langle \sqrt{vlength} \rangle$ into *VLength*. It also empties the *Result* place.

If the measurement result fulfills some predefined condition, we have found the answer we want and the computer can be stopped.

This condition is checked in the function wanted(res), which returns true if the result is what we are looking for, and false otherwise. For the example automaton, where we continue until we find the sixth data qubit in state $|1\rangle$, this function is

$$\mathrm{wanted}(res) = \begin{cases} \text{true, if } res = 1 \\ \text{false, otherwise} \end{cases} .$$

If the first measurement produced the desired result, the final result is measured. In the model, a token is taken from *AutomatonState* and put to ***FinalResult***. Also, the number of steps still to compute, in *StepsLeft*, is set to zero, which prevents *Control* from starting a new computational step.

7 Testing the model

7.1 Implementation and test run

To experimentally test the model, it was implemented using `PROD`, a reachability analysis tool for Pr/T nets [22]. The implementation contains parts which are used to compute the evolution of the computer state (*Control*, *FinishPath*, *ContinuePath* and *Computation* subsystems). *Measure* subsystem was left out to keep the amount of implementation and testing work reasonable.

The implementation handles complex numbers as four-tuples, which hold the absolute values and signs for the real and imaginary parts. The absolute values are represented as integers, scaled so that 0 is represented by zero and 1 by 10000. The operations on complex numbers (addition, multiplication, division) are written as separate C functions, which are linked together with the reachability graph generator produced by `PROD`.

The only addition to the model presented above is an additional read arc from place *StepsLeft* to transition *NextToken*. This is needed to make the transition print amplitudes (along with time step numbers) to the screen to make it easy to follow the progress of the computation. The PROD file is given in Appendix B.

Using the implemented model, a computation of 20 time steps was done, with $\Delta t = 0.25$, starting from the initial state shown in Figure 3. Paths of length 7 and shorter were used. This was expected to give a reasonable accuracy, since the mean square error of the approximated evolution operator (6.2) compared to the full U_{step} operator (6.1) decreases rapidly when $LIMIT$ grows (see Figure 15). This error is computed as

$$E_{mse} = \frac{1}{144} \sum_{r=1}^{12} \sum_{c=1}^{12} |U_{appr}(r,c) - U_{step}(r,c)|^2, \tag{7.1}$$

where (r,c) after a matrix refers to matrix elements, and the coefficient $\frac{1}{144}$ reflects the fact that the matrices are 12×12 in size.

The results given by the model were compared to the same computation done using Matlab. The Hamiltonian used in Matlab computation was built by performing the reachability analysis described in Section 5.1 both to the guard bits and to the data bits with the chosen initial state, to list all the basis states which the automaton can encounter during this specific example computation. This resulted in 36×36 Hamiltonian, which was used to compute the evolution matrix for a time step. The amplitudes computed both with the net model and with Matlab are shown in Figure 16. Although the values computed using the net model start to deviate from the correct values when the computation progresses, for short computations they are accurate enough to give a fair idea about how probability amplitudes of different states behave in relation to each other.

The effect of accuracy and number of time steps on the reachability graph size are shown in Figure 17.

The graph size grows exponentially in the path length used. This restricts the path length which can be used to 7 or at most 8 (a try with path length 8 already caused disk space problems, and thus length 7 was used in testing). Doing more computational steps also increases the reachability graph size, but only linearly. In addition, it is possible to divide the computation into several phases, saving the final state of the last step, recompiling the model and building a new reachability graph for the next step using the

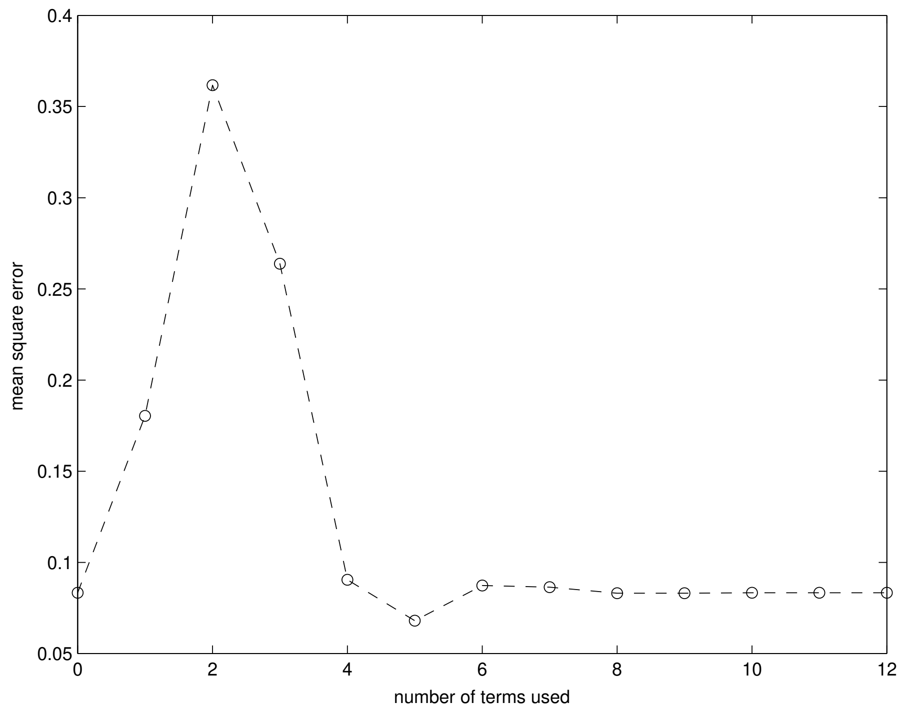


Figure 15: Effect of chosen $LIMIT$ (number of terms used to compute U_{appr}) on the error of evolution operator. X-axis shows of number of H^k-terms used in U_{appr}, and the Y-axis is the mean square error between the exactly computed U_{step} and U_{appr}.

saved state as a new initial state. This way, only the accuracy used determines the graph size. This technique was used during the testing.

Although the behavior of the automaton can be computed using the model, it is quite slow. The example computation, on an i686 processor, took about 239 hours.

7.2 Accuracy of the computation

We compute the error caused by the use of an approximated operator as

$$\begin{aligned} E &= U_{step} - U_{appr} \\ &= \sum_{i=LIMIT}^{\infty} \frac{-\mathrm{i}^k \Delta t^k}{k!} H^k. \end{aligned} \tag{7.2}$$

When an (errorless) state vector $|\psi_0\rangle$ is multiplied by an erroneous evolution operator $U_{step} + E$, can the final state be written as a sum of the correct state vector and an error vector,

$$((U_{step} + E)|\psi_0\rangle = U_{step}|\psi_0\rangle + E|\psi_0\rangle. \tag{7.3}$$

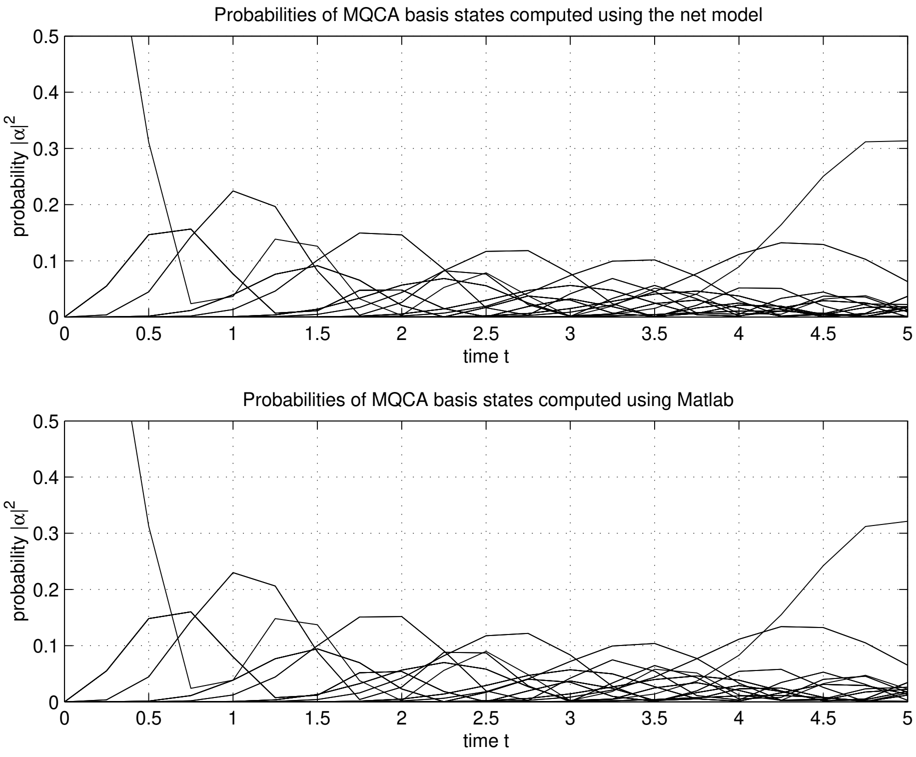


Figure 16: The probabilities of being in different basis states of MQCA, as a function of time, computed with the net model and using Matlab. Each line represents probability of one basis state. For example, the rapidly sinking line on the left is the probability of the initial state. When we start the computation, the superposed state of the automaton contains only the initial state, so probability of finding the automaton in that state is 1, and then it starts to decrease when other basis states become part of the superposition.

Although some elements in E may cancel the effect of each other during the matrix multiplication, in general elements with big magnitude in E will result in error vector $E|\psi_0\rangle$ with big magnitude. As errors in the evolution operator matrix and the state vector behave similarly, we restrict ourselves to examine the error matrix E.

In order to study the effect of different Δt on the error, we bring U_{step} and U_{appr} (for the case $LIMIT = 7$, which we used in the example computation) into closed form. Derivation of the closed form of U_{step} is shown in Appendix A, and U_{appr} can be handled similarly.

$$\begin{aligned} U_{step} = I &+ H^2(\frac{1}{16}(\cos(2\sqrt{2}\Delta t) - 1) + \frac{1}{8}(\cos(2\Delta t) - 1)) \\ &+ HL(\frac{1}{16}(\cos(2\sqrt{2}\Delta t) - 1) - \frac{1}{8}(\cos(2\Delta t) - 1)) \\ &+ \mathrm{i}H(\frac{-1}{4\sqrt{2}}\sin(2\sqrt{2}\Delta t) - \frac{1}{4}\sin(2\Delta t)) + \mathrm{i}L(\frac{-1}{4\sqrt{2}}\sin(2\sqrt{2}\Delta t) + \frac{1}{4}\sin(2\Delta t)). \end{aligned} \tag{7.4}$$

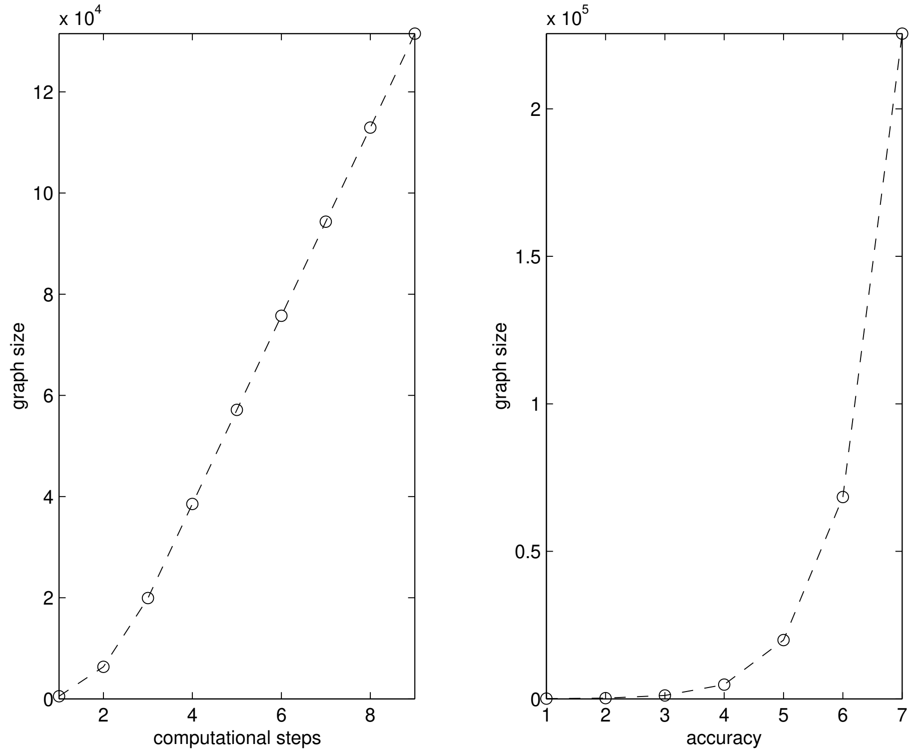


Figure 17: The effect of the number of computational steps and the accuracy (path length) used on the reachability graph size (number of nodes).

$$\begin{aligned} U_{appr} &= I + H^2\Big(-\frac{(\Delta t)^2}{2} + \frac{(\Delta t)^4}{4} - \frac{(\Delta t)^6}{18}\Big) + HL\Big(\frac{(\Delta t)^4}{12} - \frac{(\Delta t)^6}{30}\Big) \\ &\quad + \mathrm{i}H\Big(-\Delta t + (\Delta t)^3 - \frac{(\Delta t)^5}{3}\Big) + \mathrm{i}L\Big(\frac{(\Delta t)^3}{3} - \frac{(\Delta t)^5}{5}\Big) \end{aligned} \tag{7.5}$$

where L is an auxiliary matrix,

$$L = \frac{H^3 - 6H}{2}. \tag{7.6}$$

For a different $LIMIT$, the form of U_{appr} would remain similar. The matrices H, L, H^2 and HL would be exactly the same, and there would still be alternating positive and negative $(\Delta t)^k$ terms, just the number of $(\Delta t)^k$ terms would change.

By combining these, the expression for the error matrix becomes

$$
\begin{aligned}
E =& \mathrm{i}H(\Delta t - (\Delta t)^3 + \frac{(\Delta t)^5}{3} - \frac{1}{4}(\frac{1}{\sqrt{2}}\sin(2\sqrt{2}\Delta t) + \sin(2\Delta t))) \\
&+ \mathrm{i}L(-\frac{(\Delta t)^3}{3} + \frac{(\Delta t)^5}{5} - \frac{1}{4}(\frac{1}{\sqrt{2}}\sin(2\sqrt{2}\Delta t) - \sin(2\Delta t))) \\
&+ H^2(\frac{(\Delta t)^2}{2} - \frac{(\Delta t)^4}{4} + \frac{(\Delta t)^6}{18} + \frac{1}{16}(\cos(2\sqrt{2}\Delta t) + 2\cos(2\Delta t) - 3) \\
&+ HL(-\frac{(\Delta t)^4}{12} + \frac{(\Delta t)^6}{30} + \frac{1}{16}(\cos(2\sqrt{2}\Delta t) + 2\cos(2\Delta t) + 1)).
\end{aligned} \tag{7.7}
$$

When studying the effect of growing Δt, only the $(\Delta t)^k$ terms are of interest because the sin and cos terms can not grow without limit. Of the matrices H, L, H^2 and HL, only H^2 and HL have some overlapping areas of nonzero elements; in all other matrix pairs, the locations of nonzero elements are mutually nonoverlapping. This way, the effects of these matrices in the expression of E cannot completely cancel each other, and we can see that the error grows polynomially in Δt (the order of the polynom depending on $LIMIT$). This implies that the model is most suitable for studying the operation of the computer by computing state vector values at successive small Δt. Trying to find out the state of the computer after a single long evolution period would give more erroneous results.

We assumed that our Hamiltonian is errorless, and possible errors are introduced by using an approximation for the evolution operator. But, we can as well think that our Hamiltonian is an approximative one (which deviates from the correct Hamiltonian H by some error component H_{err}), and the evolution operator U_{appr} is computed from it without further errors, that is,

$$
U_{appr} = \mathrm{e}^{-\mathrm{i}H_{\mathrm{appr}}\Delta t} \tag{7.8}
$$

gives

$$
H_{\mathrm{appr}} = \frac{\mathrm{i}}{\Delta t}\ln U_{appr} = H + H_{\mathrm{err}}. \tag{7.9}
$$

It has been shown in [3] that using an erroneous Hamiltonian causes the error to grow at most quadratically in time. Quadratic growth of error was also noted when testing the net model. Figure 18 shows the mean square error of the state vector as the function of number of steps computed.

In addition to the inherent errors of the model, some error is also caused in the implementation by the inaccurate representation of complex numbers.

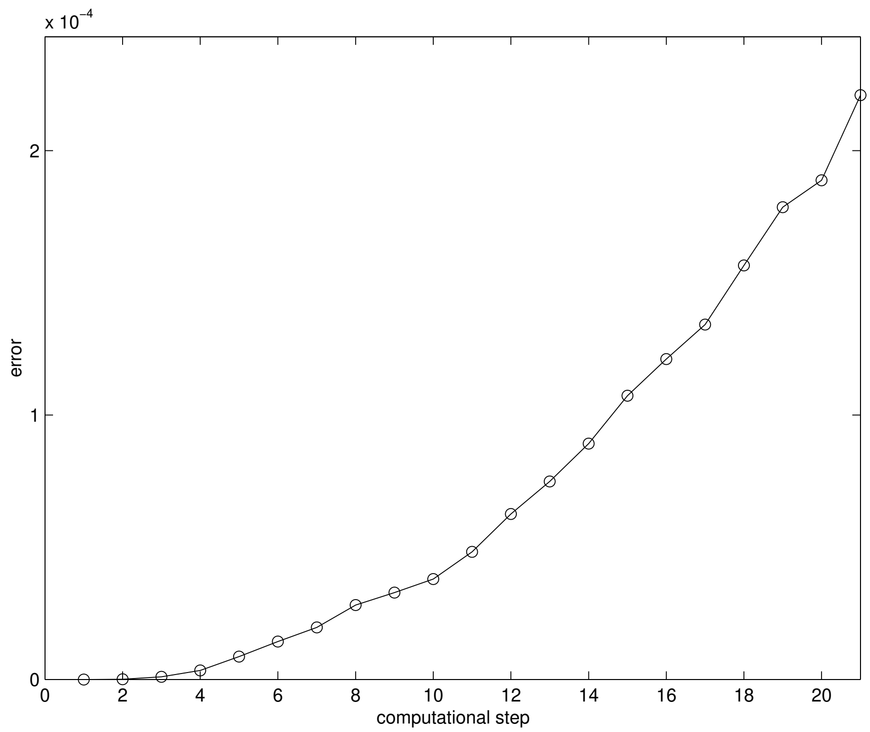


Figure 18: The growth of error (mean square error between the state vector computed using the model and U_{appr}, and the one computed with Matlab using U_{step}) as the function of the number of computational steps.

8 Conclusions

We built a model for operation of Margolus quantum cellular automaton. This was done in two parts: first, the corresponding classical automaton was modeled, and then the quantum version of it.

The model of MCA was used to analyze the behavior of the guard bit system in the classical automaton. The reachability information got from this model was put to use when modeling the MQCA.

The model of MQCA can be used to compute an approximation for the state of the chosen automaton at discrete moments of time. Also the effect of measurements (with all possible outcomes for each measurement) on the probability amplitudes can be computed, after which the computation can be continued from the collapsed state.

The main idea in the MQCA model is to make the changes in the data qubits follow those in the guard qubits. After we have analyzed the behavior of guard qubits in detail, we can easily perform computations with different data qubit values. Only the necessary data qubit configurations will be generated during the computation, so the model also gives information about reachability of the different data configurations in the MQCA.

8.1 Advantages and disadvantages

The biggest problem in the model of MQCA, compared to doing the corresponding computations with simple matrix calculations (using reduced matrices), is its slowness. The inaccuracies which accumulate in the result, due to the approximative nature of the computation and the finite accuracy used in representing the complex numbers in the implementation, can become a problem in long computations.

A 32-qubit computer, like our example automaton, is already quite big for a quantum system – it has 2^{32} basis states. Fortunately, by first modeling the corresponding classical MCA and analyzing it, we could use only a tiny fraction of these states in the final model of quantum automaton. The method we used for reducing the size of Hamiltonian matrices can make it essentially easier to mathematically analyze the behavior of Margolus quantum cellular automata, or also of other quantum computers consisting of separate data and guard qubit parts.

On the quantum side, the benefits of using this kind of net model are the possibility to represent the effect of measurements, and the insight the model gives into the details of the computation, especially the evolution of the data bit part.

8.2 Other approaches

The measurements and state collapses are handled in this work by having possible values as tokens in a place, and choosing one of the them as measurement result. The Pr/T net formalism used does not dictate which of the tokens is chosen, nor does it guarantee that the choice would be random. Therefore, we can analyze possible behaviors (the reachability graph containing information about all possible choices), but the model cannot really simulate the operation of the computer. If simulation is what we want to do, it would be better to use a stochastic formalism (like we do in [14]).

When the automaton behavior is studied in the "standard way" using matrix multiplications, the evolution step forms a kind of black box. An initial state goes in, a final state comes out, and if we want to reason what caused a certain qubit configuration to appear in the superposition or a certain amplitude to behave in an unexpected way, we will take pen and paper and start tracking down which row was multiplied by which column and what it means.

This same restriction is present in [15],[14]. There, we represent state change as one transition (where a state vector is multiplied by the evolution operator). In this work, we chose a more low-level approach. In our model, the configurations of the automaton are still qubit sequences, like they are in the automaton, and not abstract vectors like in the matrix approach. From the net model we can generate a reachability graph, which shows the progress of computation in detail, making it easier to see which data configurations are carried with which guard bit states and which paths form the amplitude of a basis state. It is not to say, however, that Petri nets would be the only (or the best) tool for accessing this type of information.

The price we have to pay for getting more details is that the computation is approximative. We cannot handle all infinitely long computational paths, so we drop away the longest ones, and thereby introduce some error into computation. The model of [14], on the other hand, is able to produce exact values.

8.3 Possible extensions

The model can be easily parameterized to perform different computations on the same type of the automaton, because the initial data configuration, length of the computational step and the block rules are set as initial marking. As long as the size of the automaton remains the same, also the behavior of the guard bit system (e.g. its initial state) can be changed relatively easily. This requires computing the reachability graph of the guard bit part anew, and giving it to the model as an initial marking. Changing the number of blocks in the automaton requires adding or removing transitions.

Margolus's goal was to study whether a quantum automaton can simulate a classical one. Therefore, all data values used in the rules are classical, just ones and zeros. It would be relatively easy to extend the model to use genuinely quantum block rules. This would require changing the $UseRuleXY$ transitions to produce two tokens with suitable amplitudes (a superposition) instead of one, and also a more complex representation for the block rules would be needed.

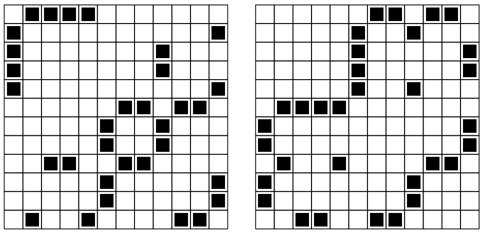


Figure 19: Matrices H (on the left) and L (on the right). Black squares denote ones, white squares zeros.

A Derivation of the closed form for the evolution operator

Now we will derive a close form representation for the evolution operator U_{step}.

We start by noticing that elements of H^3 are always 0, 2 or 6. Using H and a new matrix L(which, like H, contains only 0's and 1's), we can rewrite H^3 as

$$H^3 = 6H + 2L. \tag{A.1}$$

By straightforward matrix calculations it can be verified that

$$H^2 = L^2 \tag{A.2}$$

$$L^3 = 6L + 2H. \tag{A.3}$$

By looking at the odd powers of H, we notice a regularity in their structure: all of them contain only two nonzero values, and the two values are located at the same locations where H and L have nonzero values. It seems that, for any $n \geq 0$,

$$H^{2n+1} = a_n H + b_n L, \tag{A.4}$$

when a_n and b_n are suitably chosen. We will now show (A.4) by induction, and also find the values for coefficients a_n and b_n.

When $n = 0$, we can make (A.4) true by choosing $a_0 = 1, b_0 = 0$. Now, assuming that $H^{2n+1} = a_n H + b_n L$,

$$\begin{aligned} H^{2(n+1)+1} &= H^{(2n+1)+2} \\ &= H^{2n+1}H^2 \\ &= (a_n H + b_n L)H^2. \end{aligned} \tag{A.5}$$

Using (A.2), (A.1) and (A.3), we get

$$\begin{aligned} H^{2(n+1)+1} &= a_n H^3 + b_n L^3 \\ &= (6a_n + 2b_n)H + (2a_n + 6b_n)L. \end{aligned} \tag{A.6}$$

From (A.6) we see that (A.4) is true when we choose

$$\begin{aligned} a_{n+1} &= 6a_n + 2b_n \\ b_{n+1} &= 2a_n + 6b_n. \end{aligned} \tag{A.7}$$

After solving these recursive equations, we have

$$\begin{aligned} a_n &= 2^{2n-1}(2^n + 1) \\ b_n &= 2^{2n-1}(2^n - 1). \end{aligned} \tag{A.8}$$

Next, we will use (A.4) and (A.8) to find a closed form for $U_{step} = \mathrm{e}^{-\mathrm{i}H\Delta t}$.
From Euler's formulas

$$\cos x = \frac{1}{2}(\mathrm{e}^{\mathrm{i}x} + \mathrm{e}^{-\mathrm{i}x}), \quad \sin x = \frac{1}{2\mathrm{i}}(\mathrm{e}^{\mathrm{i}x} - \mathrm{e}^{-\mathrm{i}x}) \tag{A.9}$$

we get

$$\mathrm{e}^{-\mathrm{i}x} = \cos x - \mathrm{i}\sin x. \tag{A.10}$$

By putting (A.4) and (A.8) into the series representation of $\sin(H\Delta t)$, and reordering terms, we get

$$\begin{aligned} \sin(H\Delta t) &= \sum_{n=0}^{\infty}(-1)^n \frac{t^{2n+1}}{(2n+1)!} H^{2n+1} \\ &= (H+L)\sum_{n=0}^{\infty}(-1)^n \frac{t^{2n+1}}{(2n+1)!} 2^{3n-1} \\ &\quad + (H-L)\sum_{n=0}^{\infty}(-1)^n \frac{t^{2n+1}}{(2n+1)!} 2^{2n-1}. \end{aligned} \tag{A.11}$$

With multiplications and divisions with suitable powers of two, we can transform this into form

$$\begin{aligned} &= \frac{1}{4\sqrt{2}}(H+L)\sum_{n=0}^{\infty}(-1)^n \frac{t^{2n+1}}{(2n+1)!} 2^{3n+\frac{3}{2}} \\ &+ \frac{1}{4}(H-L)\sum_{n=0}^{\infty}(-1)^n \frac{t^{2n+1}}{(2n+1)!} 2^{2n+1}. \end{aligned} \tag{A.12}$$

The latter part we recognize as a sin series. The first part needs a bit more manipulation, namely

$$2^{3n+\frac{3}{2}} = (2^{\frac{3}{2}})^{2n+1}. \tag{A.13}$$

Now we see that

$$\sin(H\Delta t) = \frac{1}{4\sqrt{2}}(H+L)\sin(2\sqrt{2}\Delta t) + \frac{1}{4}(H-L)\sin(2\Delta t). \tag{A.14}$$

In a similar fashion, we can derive a representation for $\cos(H\Delta t)$. Since we don't have a separate formula for H^{2n}, we must use

$$\begin{aligned} H^{2n} &= HH^{2n-1} \\ &= H(a_{n-1}H + b_{n-1}L) \\ &= (2^{3n-4} + 2^{2n-3})H^2 + (2^{3n-4} - 2^{2n-3})HL. \end{aligned} \tag{A.15}$$

Since this is valid only for $n > 0$, we must separate the identity matrix I from the rest of the series form of cosine,

$$\begin{aligned}\cos(H\Delta t) = I + (H^2 + HL)\sum_{n=1}^{\infty}(-1)^n\frac{t^{2n}}{(2n)!}2^{3n-4} \\ + (H^2 - HL)\sum_{n=1}^{\infty}(-1)^n\frac{t^{2n}}{(2n)!}2^{2n-3}.\end{aligned} \tag{A.16}$$

Again, we multiply and divide by suitable coefficients, getting

$$\begin{aligned}\cos(H\Delta t) = I + \frac{1}{16}(H^2 + HL)\sum_{n=1}^{\infty}(-1)^n\frac{t^{2n}}{(2n)!}2^{3n} \\ + \frac{1}{8}(H^2 + HL)\sum_{n=1}^{\infty}(-1)^n\frac{t^{2n}}{(2n)!}2^{2n},\end{aligned} \tag{A.17}$$

which, after we have replaced 2^{3n} by $(2^{\frac{3}{2}})^{2n}$, turns out to be a sum of the identity matrix and two cosine terms,

$$\cos(H\Delta t) = I + \frac{1}{16}(H^2 + HL)(\cos(2\sqrt{2}\Delta t) - 1) + \frac{1}{8}(H^2 - HL)(\cos(2\Delta t) - 1). \tag{A.18}$$

Finally, we can combine (A.10), (A.14) and (A.18) into the desired closed-form representation for U_{step}:

$$\begin{aligned}U_{step} &= \mathrm{e}^{-\mathrm{i}H\Delta t} \\ &= \cos(H\Delta t) - \mathrm{i}\sin(H\Delta t) \\ &= \quad \frac{1}{16}(H^2 + HL)(\cos(2\sqrt{2}\Delta t) - 1) + \frac{1}{8}(H^2 - HL)(\cos(2\Delta t) - 1) \\ &\quad - \frac{\mathrm{i}}{4\sqrt{2}}(H + L)\sin(2\sqrt{2}\Delta t) - \frac{\mathrm{i}}{4}(H - L)\sin(2\Delta t) + I,\end{aligned} \tag{A.19}$$

which we can also represent in a form more convenient for further use,

$$\begin{aligned}U_{step} = I + H^2(\frac{1}{16}(\cos(2\sqrt{2}\Delta t) - 1) + \frac{1}{8}(\cos(2\Delta t) - 1)) \\ + HL(\frac{1}{16}(\cos(2\sqrt{2}\Delta t) - 1) - \frac{1}{8}(\cos(2\Delta t) - 1)) \\ + \mathrm{i}H(\frac{-1}{4\sqrt{2}}\sin(2\sqrt{2}\Delta t) - \frac{1}{4}\sin(2\Delta t)) \\ + \mathrm{i}L(\frac{-1}{4\sqrt{2}}\sin(2\sqrt{2}\Delta t) + \frac{1}{4}\sin(2\Delta t)).\end{aligned} \tag{A.20}$$

B PROD implementation of the model

B.1 PROD source file

```
#include "margQCA.src/scale.h"
#include "margQCA.src/confcmp.h"


/*
  constants to make the code more readable
*/
#define REALNUM           Mre,Sre
#define IMAGNUM           Mim,Sim
#define REALNUM_one       SCALE_UNIT, 0
#define REALNUM_zero      0,0
#define IMAGNUM_zero      0,0

#define AMPL_one           REALNUM_one, REALNUM_zero
#define AMPL_IN1           Mre1, Sre1, Mim1, Sim1
#define AMPL_IN2           Mre2, Sre2, Mim2, Sim2
#define AMPL_IN3           Mre3, Sre3, Mim3, Sim3
#define AMPL_OUT           Mre_out, Sre_out, Mim_out, Sim_out
#define ptr_AMPL_OUT       &Mre_out, &Sre_out, &Mim_out, &Sim_out

#define MINUS_I           0,0,SCALE_UNIT,1
#define PLENGTH(x)        (x*SCALE_UNIT), 0, 0, 0

#define RULEIN            in_ul, in_ur, in_ll, in_lr
#define RULEOUT           out_ul, out_ur, out_ll, out_lr
#define RULETUPLE         <.RULEIN, RULEOUT.>


#define CONF_1  c1_c1, c1_c2, c1_c3, c1_c4, c1_c5, c1_c6,
                c1_c7, c1_c8, c1_c9, c1_c10, c1_c11, c1_c12,
                c1_c13, c1_c14, c1_c15, c1_c16
#define CONF_2  c2_c1, c2_c2, c2_c3, c2_c4, c2_c5, c2_c6,
                c2_c7, c2_c8, c2_c9, c2_c10, c2_c11, c2_c12,
                c2_c13, c2_c14, c2_c15, c2_c16
#define CONF_IN cin_11, cin_12, cin_13, cin_14, cin_21, cin_22,
                cin_23, cin_24, cin_31, cin_32, cin_33, cin_34,
                cin_41, cin_42, cin_43, cin_44


#define RuleId(x) (x)
```

```
/*
  default values for accuracy, number of time steps,
  time step length and initial state of MQCA
*/

#ifndef PLENLIMIT
#define PLENLIMIT      3
#endif

#ifndef HOWMANYSTEPS
#define HOWMANYSTEPS        2
#endif

#ifndef DELTAT
/* 1.0 would be one SCALE_UNIT (see scale.h) */
#define DELTAT              2500
#endif

#ifndef HAS_automstate
#define STARTSTATE          12
#define INITCONF            1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1
#place  main_automatonstate mk(<.1, STARTSTATE, AMPL_one, INITCONF, 0.>)
#place  main_readycount     mk(<.STARTSTATE,1.>+<.1..11,0.>)
#endif

#place param_stepsleft       mk(<.HOWMANYSTEPS.>)
#place param_deltat          mk(<.DELTAT, 0, 0, 0.>)



/*
  block rules used
*/

#place param_rules mk(
     <.0,0,0,0, 0,0,0,0.>+

     <.1,0,0,0, 0,0,0,1.>+     /* one "particle" moves */
     <.0,1,0,0, 0,0,1,0.>+
     <.0,0,1,0, 0,1,0,0.>+
     <.0,0,0,1, 1,0,0,0.>+

     <.1,0,0,1, 0,1,1,0.>+     /* two "particles" meet */
     <.0,1,1,0, 1,0,0,1.>+
```

```
     <.1,1,0,0, 1,1,0,0.>+
     <.0,0,1,1, 0,0,1,1.>+
     <.1,0,1,0, 1,0,1,0.>+
     <.0,1,0,1, 0,1,0,1.>+

     <.0,1,1,1, 0,1,1,1.>+
     <.1,0,1,1, 1,0,1,1.>+
     <.1,1,0,1, 1,1,0,1.>+
     <.1,1,1,0, 1,1,1,0.>+

     <.1,1,1,1, 1,1,1,1.>)




/*
  the Hamiltonian of the automaton
*/

#define NUMOFSTATES 12

#place param_numofstates mk(<.NUMOFSTATES.>)

#place param_numsuccessors mk(
     <.1,4.> + <.2,2.> + <.3,2.> + <.4,2.> +
     <.5,2.> + <.6,4.> + <.7,2.> + <.8,2.> +
     <.9,4.> + <.10,2.>+ <.11,2.>+ <.12,4.>)

#place param_successors mk(
     <.12,2, RuleId(11),1.> + <.12,5, RuleId(13),2.> +
     <.12,10,RuleId(24),3.> + <.12,11,RuleId(22),4.> +
     <.2,12, RuleId(11),1.> + <.2,1,  RuleId(13),2.> +
     <.5,12, RuleId(13),1.> + <.5,1,  RuleId(11),2.> +
     <.1,2,  RuleId(13),1.> + <.1,5,  RuleId(11),2.> +
     <.1,3,  RuleId(42),3.> + <.1,4,  RuleId(44),4.> +
     <.3,1,  RuleId(42),1.> + <.3,9,  RuleId(44),2.> +
     <.4,1,  RuleId(44),1.> + <.4,9,  RuleId(42),2.> +
     <.9,3,  RuleId(44),1.> + <.9,4,  RuleId(42),2.> +
     <.9,7,  RuleId(31),3.> + <.9,8,  RuleId(33),4.> +
     <.7,9,  RuleId(31),1.> + <.7,6,  RuleId(33),2.> +
     <.8,9,  RuleId(33),1.> + <.8,6,  RuleId(31),2.> +
     <.6,7,  RuleId(33),1.> + <.6,8,  RuleId(31),2.> +
     <.6,10, RuleId(22),3.> + <.6,11, RuleId(24),4.> +
     <.10,6, RuleId(22),1.> + <.10,12,RuleId(24),2.> +
     <.11,6, RuleId(24),1.> + <.11,12,RuleId(22),2.>)
```

```
/*
  places used by all subsystems
*/

#place main_cursor           mk(<.0.>)
#place main_notreadycount mk(<.0.>)
#place main_nextstate
#place main_vectorlength


/*
  auxiliary places for different subsystems
*/

#place continue_tmpvalues       mk(<.0,0.>)

#place finish_iterstate       mk(<.1.>)
#place finish_newpath

#place control_iterstate       mk(<.1.>)
#place control_itermoved       mk(<.0.>)




/* ------------------------------------------------------------
   transition for skipping measurements
   ------------------------------------------------------------ */

#trans main_nomeasure
in {
   main_notreadycount:  <.0.>;

   main_cursor:         <.0.>;
   }
out {
   main_notreadycount:  <.0.>;

   main_vectorlength:   <.REALNUM_one.>;
   }

#endtr
```

```
/* ------------------------------------------------------------
   subsystem CONTINUE
   ------------------------------------------------------------ */


#trans continue_startstep
in {
   main_notreadycount:    <.qnum.>;
   main_automatonstate:   <.qnum, fromstate, AMPL_IN1, CONF_IN, plength.>;
   main_cursor:           <.0.>;
   param_numsuccessors:   <.fromstate, numsuccstates.>;
   continue_tmpvalues:    <.garbage1, garbage2.>;
}

gate (plength > 0) && (plength < PLENLIMIT);

out {
   main_notreadycount: <.qnum+numsuccstates.>;
   main_automatonstate:<.qnum, fromstate, AMPL_IN1, CONF_IN, PLENLIMIT.>;
   main_cursor:        <.numsuccstates.>;
   param_numsuccessors:<.fromstate, numsuccstates.>;
   continue_tmpvalues: <.qnum, plength.>;
}

#endtr




#trans continue_handlesucc
in {
  main_cursor:        <.cnt.>;
  main_automatonstate:<.oldqn,fromstate, AMPL_IN1, CONF_IN, PLENLIMIT.>;
  continue_tmpvalues: <.oldqn, plen.>;
  param_successors:   <.fromstate, tostate, ruleid, cnt.>;
  }
gate (cnt > 0);
out {
  main_cursor:        <.cnt-1.>;
  main_automatonstate:<.oldqn,fromstate, AMPL_IN1, CONF_IN, PLENLIMIT.>;
  continue_tmpvalues: <.oldqn, plen.>;
  param_successors:   <.fromstate, tostate, ruleid, cnt.>;

  main_nextstate:     <.tostate,ruleid,AMPL_IN1,CONF_IN,plen,oldqn+cnt.>;
  }
```

```
#endtr



/* ------------------------------------------------------------
   subsystem COMPUTATION
   ------------------------------------------------------------
   all transitions are folded into one
*/


#trans computation_useruleXY
in {
  main_notreadycount: <.qnum.>;
  param_rules:        <.in1, in2, in3, in4,  out1, out2, out3, out4.>;
  main_cursor:        <.0.>;
  param_deltat:       <.dt, 0, 0, 0.>;

  main_nextstate:     <.tostate, rule, AMPL_IN1,CONF_IN, plength, qnum.>;
  }

gate (
(rule==RuleId(11) && (in1 == cin_11) && (in2 == cin_12) &&
                     (in3 == cin_21) && (in4 == cin_22))||
(rule==RuleId(13) && (in1 == cin_13) && (in2 == cin_14) &&
                     (in3 == cin_23) && (in4 == cin_24))||
(rule==RuleId(31) && (in1 == cin_31) && (in2 == cin_32) &&
                     (in3 == cin_41) && (in4 == cin_42))||
(rule==RuleId(33) && (in1 == cin_33) && (in2 == cin_34) &&
                     (in3 == cin_43) && (in4 == cin_44))||
(rule==RuleId(22) && (in1 == cin_22) && (in2 == cin_23) &&
                     (in3 == cin_32) && (in4 == cin_33))||
(rule==RuleId(24) && (in1 == cin_24) && (in2 == cin_21) &&
                     (in3 == cin_34) && (in4 == cin_31))||
(rule==RuleId(42) && (in1 == cin_42) && (in2 == cin_43) &&
                     (in3 == cin_12) && (in4 == cin_13))||
(rule==RuleId(44) && (in1 == cin_44) && (in2 == cin_41) &&
                     (in3 == cin_14) && (in4 == cin_11)));

out {
   main_notreadycount:   <.qnum.>;
   param_rules:          <.in1, in2, in3, in4,
                                out1, out2, out3, out4.>;
   main_cursor:          <.0.>;
   param_deltat:         <.dt, 0, 0, 0.>;
```

```
main_automatonstate:
     (rule==RuleId(11))<.qnum,tostate, AMPL_OUT,
               out1,  out2,  cin_13, cin_14,
               out3,  out4,  cin_23, cin_24,
               cin_31, cin_32, cin_33, cin_34,
               cin_41, cin_42, cin_43, cin_44,
               plength+1.>+
     (rule==RuleId(13))<.qnum,tostate, AMPL_OUT,
               cin_11, cin_12, out13, out14,
               cin_21, cin_22, out23, out24,
               cin_31, cin_32, cin_33, cin_34,
               cin_41, cin_42, cin_43, cin_44,
               plength+1.>+
     (rule==RuleId(31))<.qnum,tostate, AMPL_OUT,
               cin_11, cin_12, cin_13, cin_14,
               cin_21, cin_22, cin_23, cin_24,
               out1,  out2,  cin_33, cin_34,
               out3,  out4,  cin_43, cin_44,
               plength+1.>+
      (rule==RuleId(33))<.qnum,tostate, AMPL_OUT,
               cin_11, cin_12, cin_13, cin_14,
               cin_21, cin_22, cin_23, cin_24,
               cin_31, cin_32, out1, out2,
               cin_41, cin_42, out3, out4,
               plength+1.>+
     (rule==RuleId(22))<.qnum,tostate, AMPL_OUT,
               cin_11, cin_12, cin_13, cin_14,
               cin_21, out1,  out2,  cin_24,
               cin_31, out3,  out4,  cin_34,
               cin_41, cin_42, cin_43, cin_44,
               plength+1.>+
     (rule==RuleId(24))<.qnum,tostate, AMPL_OUT,
               cin_11, cin_12, cin_13, cin_14,
               out2,  cin_22, cin_23, out1,
               out4,  cin_32, cin_33, out3,
               cin_41, cin_42, cin_43, cin_44,
               plength+1.>+
     (rule==RuleId(42))<.qnum,tostate, AMPL_OUT,
               cin_11, out3,  out4,  cin_14,
               cin_21, cin_22, cin_23, cin_24,
               cin_31, cin_32, cin_33, cin_34,
               cin_41, out1,  out2,  cin_44,
               plength+1.>+
     (rule==RuleId(44))<.qnum,tostate, AMPL_OUT,
               out4, cin_12, cin_13, out3,
```

```
                 cin_21, cin_22, cin_23, cin_24,
                 cin_31, cin_32, cin_33, cin_34,
                 out2, cin_42, cin_43, out1,
                 plength+1.>;
   }

comp {
   /* computation of new amplitude
               (multiplication by constant/pathlength)*/
   scaletimes(AMPL_IN1, MINUS_I, ptr_AMPL_OUT);
   scaletimes(AMPL_OUT, dt, 0, 0, 0, ptr_AMPL_OUT);
   scaledivide(AMPL_OUT, PLENGTH(plength), ptr_AMPL_OUT);
   Accept();
   }
#endtr




/* ------------------------------------------------------------
   subsystem CONTROL
   ------------------------------------------------------------ */


#trans control_nexttoken
in {
   main_readycount:      <.state, ready.>;
   main_notreadycount:   <.notready.>;
   main_automatonstate:  <.moved+1,state, AMPL_IN1, CONF_1, 0.>;
   param_stepsleft:      <.stepsleft.>;
   control_iterstate:    <.state.>;
   control_itermoved:    <.moved.>;
   main_vectorlength:    <.REALNUM.>;
   }

gate (moved < ready)  && (state>0);

out {
   main_readycount:      <.state, ready.>;
   main_notreadycount:   <.notready+1.>;
   main_automatonstate:  <.notready+1,state, AMPL_OUT, CONF_1, 1.>;
   param_stepsleft:      <.stepsleft.>;
   control_iterstate:    <.state.>;
   control_itermoved:    <.moved+1.>;
   main_vectorlength:    <.REALNUM.>;
```

```
   }
comp {
   /* adjust vector length if needed */
   scaledivide(AMPL_IN1, REALNUM, IMAGNUM_zero, ptr_AMPL_OUT);

   /* print current situation to the screen */
   printf("\nafter step %ld: guardstate %2ld,
             ampl(%c%c%c%c %c%c%c%c %c%c%c%c %c%c%c%c)
             == %c%1.4f %c %1.4f i",
   HOWMANYSTEPS-stepsleft, state,
   (c1_c1==0)?'-':'x', (c1_c2==0)?'-':'x', (c1_c3==0)?'-':'x',
      (c1_c4==0)?'-':'x', (c1_c5==0)?'-':'x', (c1_c6==0)?'-':'x',
      (c1_c7==0)?'-':'x', (c1_c8==0)?'-':'x', (c1_c9==0)?'-':'x',
      (c1_c10==0)?'-':'x', (c1_c11==0)?'-':'x', (c1_c12==0)?'-':'x',
   (c1_c13==0)?'-':'x', (c1_c14==0)?'-':'x', (c1_c15==0)?'-':'x',
   (c1_c16==0)?'-':'x',
   (Sre1>0)?'-':' ',(double)Mre1/(double)SCALE_UNIT,
   (Sim1>0)?'-':'+',(double)Mim1/(double)SCALE_UNIT);

   Accept();
   }
#endtr



#trans control_nextstate
in {
   control_iterstate:    <.state.>;
   control_itermoved:    <.ready.>;
   main_readycount:      <.state, ready.>;
   param_numofstates:    <.maxstate.>;
   main_vectorlength:    <.REALNUM.>;
   }

gate (state <= maxstate) && (state>0);

out {
   control_iterstate:    <.state+1.>;
   control_itermoved:    <.0.>;
   main_readycount:      <.state, 0.>;
   param_numofstates:    <.maxstate.>;
   main_vectorlength:    <.REALNUM.>;
   }
#endtr
```

```
#trans control_nextstep
in {
   param_stepsleft:      <.stepsleft.>;
   control_iterstate:    <.maxstate+1.>;
   param_numofstates:    <.maxstate.>;

   main_vectorlength:    <.REALNUM.>;
   }

gate (stepsleft>0);

out {
   param_stepsleft:      <.stepsleft-1.>;
   control_iterstate:    <.1.>;
   param_numofstates:    <.maxstate.>;

    main_cursor:         <.0.>;
}
#endtr




/* -------------------------------------------------------------
   subsystem CONTROL
   ------------------------------------------------------------- */


#trans finish_startcompare
in {
   main_notreadycount:    <.qnum.>;

   main_automatonstate:   <.qnum,state, AMPL_IN1, CONF_IN, PLENLIMIT.>;
   main_cursor:           <.0.>;
   }
out {
   main_notreadycount:    <.qnum-1.>;

   finish_newpath:        <.state, AMPL_IN1, CONF_IN.>;
   }
#endtr
```

```
#trans finish_iterempty
in {
   main_readycount:      <.state, 0.>;

   finish_newpath:       <.state, AMPL_IN1, CONF_IN.>;
   }
out {
   main_readycount:      <.state, 1.>;

   main_automatonstate:  <.1, state, AMPL_IN1, CONF_IN, 0.>;
   main_cursor:          <.0.>;
   }
#endtr



#trans finish_iternoteq
in {
   main_automatonstate:  <.confcount,state, AMPL_IN1, CONF_1, 0.>;
   finish_newpath:       <.state, AMPL_IN2, CONF_2.>;
   main_readycount:      <.state, maxcount.>;
   finish_auxcurrent:    <.confcount.>;
   }

gate (! confeq(CONF_1, CONF_2) && (confcount  < maxcount);

out {
   main_automatonstate:  <.confcount,state, AMPL_IN1, CONF_1, 0.>;
   finish_newpath:       <.state, AMPL_IN2, CONF_2.>;
   main_readycount:      <.state, maxcount.>;
   finish_auxcurrent:    <.confcount+1.>;
   }
#endtr



#trans finish_iterstop
in {
   main_automatonstate:  <.maxcount,state, AMPL_IN1, CONF_1, 0.>;
   main_readycount:      <.state, maxcount.>;
   finish_auxcurrent:    <.maxcount.>;

   finish_newpath:       <.state, AMPL_IN2, CONF_2.>;
   }
gate (! confeq(CONF_1, CONF_2));
out {
```

```
   main_automatonstate:  <.maxcount,state, AMPL_IN1, CONF_1, 0.> +
                         <.maxcount+1,state, AMPL_IN2, CONF_2,0.>;
   main_readycount:      <.state, maxcount+1.>;
   finish_auxcurrent:    <.1.>;

   main_cursor:          <.0.>;
   }
#endtr




#trans finish_itereq
in {
   main_automatonstate:  <.confcount,state, AMPL_IN1, CONF_IN, 0.>;
   main_readycount:      <.state, maxcount.>;
   finish_auxcurrent:    <.confcount.>;

   finish_newpath:       <.state, AMPL_IN2, CONF_IN.>;
   }

gate (confcount <= maxcount);

out {
   main_automatonstate:  <.confcount,state, AMPL_OUT, CONF_IN, 0.>;
   main_readycount:      <.state, maxcount.>;
   finish_auxcurrent:    <.1.>;

   main_cursor:          <.0.>;
   }
comp {
   /* combine the amplitudes */
   scaleplus(AMPL_IN1, AMPL_IN2, ptr_AMPL_OUT);
   Accept();
   }
#endtr
```

B.2 Auxiliary C files

B.2.1 confcmp.h

```
#ifndef CONFCMP_H
#define CONFCMP_H

#define CONFPARAM1 unsigned long c1_c1, unsigned long c1_c2,
unsigned long c1_c3, unsigned long c1_c4, unsigned long c1_c5,
unsigned long c1_c6, unsigned long c1_c7, unsigned long c1_c8,
```

```
unsigned long c1_c9, unsigned long c1_c10, unsigned long c1_c11,
unsigned long c1_c12, unsigned long c1_c13, unsigned long c1_c14,
unsigned long c1_c15, unsigned long c1_c16

#define CONFPARAM2 unsigned long c2_c1, unsigned long c2_c2,
unsigned long c2_c3, unsigned long c2_c4, unsigned long c2_c5,
unsigned long c2_c6, unsigned long c2_c7, unsigned long c2_c8,
unsigned long c2_c9, unsigned long c2_c10, unsigned long c2_c11,
unsigned long c2_c12, unsigned long c2_c13, unsigned long c2_c14,
unsigned long c2_c15, unsigned long c2_c16

int confeq(CONFPARAM1, CONFPARAM2);


#endif
```

B.2.2 confcmp.c

```
#include "confcmp.h"

int confeq(CONFPARAM1, CONFPARAM2)
{
return (
 (c1_c1 == c2_c1) &&
 (c1_c2 == c2_c2) &&
 (c1_c3 == c2_c3) &&
 (c1_c4 == c2_c4) &&
 (c1_c5 == c2_c5) &&
 (c1_c6 == c2_c6) &&
 (c1_c7 == c2_c7) &&
 (c1_c8 == c2_c8) &&
 (c1_c9 == c2_c9) &&
 (c1_c10 == c2_c10) &&
 (c1_c11 == c2_c11) &&
 (c1_c12 == c2_c12) &&
 (c1_c13 == c2_c13) &&
 (c1_c14 == c2_c14) &&
 (c1_c15 == c2_c15) &&
 (c1_c16 == c2_c16));
}
```

B.2.3 scale.h

```
#ifndef SCALE_H
#define SCALE_H
```

```
#define SCALE_UNIT 10000   /* number 1.0 */

#define CPLX_IN1 unsigned long Mre1,unsigned long Sre1,
                 unsigned long Mim1,unsigned long Sim1
#define CPLX_IN2 unsigned long Mre2,unsigned long Sre2,
                 unsigned long Mim2,unsigned long Sim2
#define CPLX_IN_res unsigned long *Mre_res,unsigned long *Sre_res,
                    unsigned long *Mim_res,unsigned long *Sim_res

void scaleplus(CPLX_IN1, CPLX_IN2, CPLX_IN_res);
void scaletimes(CPLX_IN1, CPLX_IN2, CPLX_IN_res);
void scaledivide(CPLX_IN1, CPLX_IN2, CPLX_IN_res);

#endif
```

B.2.4 scale.c

```
#include "scale.h"
#include <stdio.h>

void scaleplus(CPLX_IN1, CPLX_IN2, CPLX_IN_res)
{
  /*  REAL part

    simply a + b or -(a + b),  if the signs are the same

    a - b    if a>b, a>0, b<0
   -(a - b)  if a>b, a<0, b>0

    b - a    if b>a, b>0, a<0
   -(b - a)  if b>a, b<0, a>0
  */

  if (Sre1 == Sre2)
    {
      *Mre_res = Mre1 + Mre2;
      *Sre_res = Sre1;
    }
  else
  if (Mre1 > Mre2)
    {
      *Mre_res = Mre1 - Mre2;
      if (Sre1 != 0) *Sre_res = 1; else *Sre_res = 0;
    }
  else /* Mre2 > Mre1 */
    {
```

```
      *Mre_res = Mre2 - Mre1;
      if (Sre2 != 0) *Sre_res = 1; else *Sre_res = 0;
    }

  /*
    IMAG part (just the same as above but for im-variables)
  */

  if (Sim1 == Sim2)
    {
      *Mim_res = Mim1 + Mim2;
      *Sim_res = Sim1;
    }
  else
  if (Mim1 > Mim2)
    {
      *Mim_res = Mim1 - Mim2;
      if (Sim1 != 0) *Sim_res = 1; else *Sim_res = 0;
    }
  else /* Mim2 > Mim1 */
    {
      *Mim_res = Mim2 - Mim1;
      if (Sim2 != 0) *Sim_res = 1; else *Sim_res = 0;
    }

   /*
    Always represent zero as +0
   */
  if (*Mre_res == 0) *Sre_res = 0;
  if (*Mim_res == 0) *Sim_res = 0;
}


void scaledivide(CPLX_IN1, CPLX_IN2, CPLX_IN_res)
{
  /*
    division by pure imaginary number
  */
  if ((Mre2 == 0) && (Mim2 != 0))
    {
      Sim2=0;
      printf("\nDivision by ib not implemented\n");
      exit(1);
    }
  else
    /*
```

```
      division by real number
     */
  if ((Mre2 != 0) && (Mim2 == 0))
    {
      *Mre_res = (Mre1 * SCALE_UNIT) / Mre2;
      *Mim_res = (Mim1 * SCALE_UNIT) / Mre2;
      if (Sre1 == Sre2) *Sre_res=0; else *Sre_res=1;
      if (Sim1 == Sre2) *Sim_res=0; else *Sim_res=1;
    }
  else
    /*
      not needed here
    */
    {
      printf("\nDivision by a+ib not implemented\n");
      exit(1);
    }
}


void scaletimes(CPLX_IN1, CPLX_IN2, CPLX_IN_res)
{
  unsigned long Mre_tmp1, Sre_tmp1, Mre_tmp2, Sre_tmp2;
  unsigned long Mim_tmp1, Sim_tmp1, Mim_tmp2, Sim_tmp2;

  /* new real part: re1 * re2 - im1 * im2
     both M1 and M2 are multiplied by SCALE_UNIT; the result needs to be
     multiplied by it only once, whence the division
   */
  Mre_tmp1 = (Mre1 * Mre2) / SCALE_UNIT;
  Sre_tmp1 = (Sre1 == Sre2)?0:1;

  Mre_tmp2 = (Mim1 * Mim2) / SCALE_UNIT;
  Sre_tmp2 = (Sim1 == Sim2)?1:0; /* i^2 changes sign */

  /* new imaginary part: im1 * re2 + re1 * im2 */
  Mim_tmp1 = (Mim1 * Mre2) / SCALE_UNIT;
  Sim_tmp1 = (Sim1 == Sre2)?0:1;

  Mim_tmp2 = (Mre1 * Mim2) / SCALE_UNIT;
  Sim_tmp2 = (Sre1 == Sim2)?0:1;

  scaleplus(Mre_tmp1, Sre_tmp1,  Mim_tmp1, Sim_tmp1,
    Mre_tmp2, Sre_tmp2,  Mim_tmp2, Sim_tmp2,
    Mre_res,  Sre_res,   Mim_res,  Sim_res);
}
```

References

[1] Paul Benioff. The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. *Journal of Statistical Physics*, 22:495–507, 1981.

[2] Michael Biafore. *A computational role for the one-dimensional Lieb-Shultz-Mattis spin model*. PhD thesis, Massachusetts Institute of Technology, May 1994.

[3] Scott H. Clearwater Colin P. Williams. *Explorations in Quantum Computing*. Springer-Verlag, New York, 1998.

[4] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6/7):467–488, 1982.

[5] Richard P. Feynman. Quantum mechanical computers. *Optics News*, 11:11–20, February 1985.

[6] E. Fredkin and T. Toffoli. Conservative logic. In E. Fredkin, R. Landauer, and T. Toffoli, editors, *Proceedings of the Physics of Computation Conference*, pages 219–253, 1982.

[7] Hartmann J. Genrich. Predicate/Transition nets. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency, Part I*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, Berlin, 1987.

[8] Tino Gramß. The theory of quantum computation: An introduction. In Tino Gramß, Stefan Bornholdt, Michael Groß, Melanie Mitchell, and Thomas Pellizzari, editors, *Non-Standard Computation*, pages 141–178. Wiley-VCH, Weinheim, 1998.

[9] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of 28th ACM STOC*, pages 212–219, 1996.

[10] Jozef Gruska. *Quantum Computing*. Mc Graw-Hill, London, 1999.

[11] Seth Lloyd. Universal quantum simulators. *Science*, 273(23):1073–1078, August 1996.

[12] Norman Margolus. Quantum computation. *Annals of the New York Academy of Sciences*, 480:487–497, 1986.

[13] Norman Margolus. Parallel quantum computation. In W.H.Zurek, editor, *Complexity, Entropy, and the Physics of Information*, volume VIII of *SFI Studies in the Sciences of Complexity*, pages 273–287. Addison-Wesley, 1990.

[14] Leo Ojala, Elina Parviainen, Olli-Matti Penttinen, Teemu Tynjälä, and Harriet Beaver. Modeling Feynman's quantum computer using stochastic high-level Petri nets. In *Proceedings of 2001 IEEE International Conference on Systems, Man & Cybernetics (CD-ROM)*, pages 2735–2741, 2001.

[15] Leo Ojala, Teemu Tynjälä, and Harriet Beaver. Modeling serial quantum processors using high-level Petri nets. In *Proceedings of the 5th World Multiconference on Systemics, Cybernetics and Informatics - SCI'2001*, 2001.

[16] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, University of Bonn, 1962.

[17] Arthur O. Pittenger. *Introduction to quantum computing algorithms.* Birkhäuser, Boston, 2000.

[18] John Preskill. Lecture notes on quantum computation, 1998. Available at: <url:http://theory.caltech.edu/~preskill/ph229/#lecture>.

[19] Palash Sarkar. A brief history of cellular automata. *ACM Computing Surveys*, 32(1):80–107, 2000.

[20] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[21] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A New Environment for Modeling.* MIT Press, Cambridge, MA, 1987.

[22] Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, and Tino Pyssysalo. Prod reference manual. Technical Report B 13, Helsinki University of Technology, August 1995.

[23] John von Neumann and Arthur W. Burks, editors. *Theory of self-reproducing automata.* University of Illinois Press, Champaign, IL, 1966.

[24] John Watrous. On one-dimensional cellular automata. In *Proceedings of 36th IEEE FOCS*, pages 528–537, 1995.

[25] S. Wolfram. *Theory and applications of cellular automata: including selected papers 1983-1986.* World Scientific Publishing Co., Inc., River Edge, NJ, 1986.

[26] Christof Zalka. Efficient simulation of quantum systems by quantum computers. *Proc. Roy. Soc.of London A*, 454:313–322, 1998.