Distributed Quantum Programming

Ellie D'Hondt Yves Vandriessche

Software Languages Lab
Vrije Universiteit Brussel

10F719, Pleinlaan 2, 1050 Elsene, Belgium
tel. +32 629 11 05, fax. +32 629 35 25
{Ellie.DHondt, Yves.Vandriessche}@vub.ac.be

June 15, 2012

Abstract

In this paper we explore the structure and applicability of the Distributed Measurement Calculus (DMC), an assembly language for distributed measurement-based quantum computations. We describe the formal language's syntax and semantics, both operational and denotational, and state several properties that are crucial to the practical usability of our language, such as equivalence of our semantics, as well as compositionality and context-freeness of DMC programs. We show how to put these properties to use by constructing a composite program that implements distributed controlled operations, in the knowledge that the semantics of this program does not change under the various composition operations. Our formal model is the basis of a quantum virtual machine construction for distributed quantum computations, which we elaborate upon in the latter part of this work. This virtual machine embodies the formal semantics of DMC such that programming execution no longer needs to be analysed by hand. Far from a literal translation, it requires a substantial concretisation of the formal model at the level of data structures, naming conventions and abstraction mechanisms. At the same time we provide automatisation techniques for program specification where possible to obtain an expressive and user-friendly programming environment.

Keywords: quantum computing, measurement-based quantum computing, distributed computing, formal models, virtual machines.

1 Introduction

During the last decennia, quantum information has managed to become a significant field of research in the exact and applied sciences. Although it is a relatively new discipline one can currently discern several sub-disciplines such as quantum cryptography, information theory, computability, error correction, fault tolerance, computations and of course there is also the experimental research on the construction of quantum computers (Nielsen and Chuang, 2000). Nevertheless, the development of quantum information as a proper computational domain of computer science is lagging behind. Indeed there is no such thing as a quantum computational paradigm. By this we mean a framework in which quantum problems can be expressed and solved in terms of data structures, algorithms, techniques such as abstraction, all of these wrapped up in an associated programming language. Paradigm building has proved to be an extremely useful approach in computer science. It has led to theoretically equal but practically very different programming frameworks, such as functional, imperative, logic and object-oriented programming. For this reason we expect this approach to be crucial also in developing quantum programming paradigms.

The first step in paradigm building is to construct a low-level quantum programming language and determine its properties. By low-level we mean that we need to define syntactical expressions for each physically allowed quantum operation: preparation, unitary transformation, measurement, combined with classical control expressions if so desired. The syntax in its own is not the goal, but rather a means by which to facilitate investigations with computer science techniques. First, we need to determine the functionality of a quantum program, its *semantics*. The most obvious way to do this is the operational semantics, the most practical is the denotational semantics. While in the former a program's meaning is given as a sequence of state-changing operations, in the latter it is instead a mathematical object. Paramount is linking both, so that one can use whichever in future analyses. Through a programming language's semantics one can investigate notions such as composition and context-freeness. These properties are crucial when one wants to build more complex programs. Indeed, they allow the semantics of these larger programs to be built up from that of smaller components using rules for composition, rather than having to be determined from scratch. While these properties may seem obvious, computer science is littered with examples where they were mistakenly taken to be true, leading to problems in programming language development (see for example (Brock and Ackerman, 1981)).

Recent advances in quantum communication and cryptographic computations motivate the need for a programming paradigm centred on *distributed* quantum computations. In a distributed system one has concurrently acting agents in separated locations, operating locally on a quantum state, which may be entangled over agents, and coordinating their actions via communication. Formal frameworks for distributed quantum computation have only very recently begun to appear. Typically, these are a combination of classical process theory, which formalises notions of concurrency and communication, the quantum circuit model, i.e. local operations are unitary transformations of an agent's qubits, and given initial shared entanglement. First, there is the work in (Lalire and Jorrand, 2004; Jorrand and Lalire, 2005), which is directly built upon classical process calculi. While this model profits from being closely related to existing classical models, the disadvantage is that it is hard to focus on quantum behaviour. A different approach is advanced in the model known as *communicating quantum processes* or CQP (Gay and Nagarajan, 2004; Gay *et al.*, 2005), where the typical communication primitives of process calculi are combined with computational primitives from QPL (Selinger, 2003). The basic model of CQP is more transparent. This model serves as a basis for the development of formal verification techniques, in particular for proving the security of larger scale quantum communication systems. In related work, a probabilistic model-checking tool built upon an existing automated verification component is developed (Gay *et al.*, 2005). There is also the work in (Adão and Mateus, 2005), which is specifically tailored to security issues in cryptographic protocols. In our work, however, we are much more interested in the expressiveness of quantum distributed computations and the behavioural properties of distributed primitives. In a way, we take the inverse approach, assuming that computations are well-defined and investigating what programming concepts are at work, instead of the other way around.

While the fact that formal verification tools for distributed quantum computation are currently under development may suggest that a mature distributed paradigm already exists, this is actually not the case. Distributed protocols are still very much conceived on intuition, and there is no good notion or formalisation of the programming concepts that are at work. We therefore require adequate formal tools with which to explore and evolve the distributed quantum computing paradigm. In this paper we explore the structure and applicability of the Distributed Measurement Calculus (DMC) (D'Hondt, 2005; Danos *et al.*, 2005), an assembly language for distributed measurement-based quantum computations. We describe its syntax and semantics, both operational and denotational, and state several properties that are crucial to the practical usability of our language, such as equivalence of our semantics, as well as compositionality and context-freeness of DMC programs. We show how to put these properties to use by constructing a composite program that implements distributed controlled operations, and demonstrate that the semantics of this program does not change under the various composition operations. Finally, we elaborate on an implementation for the DMC language which we developed under the form of a virtual machine, a platform-independent programming environment that abstracts away details of the underlying hardware or operating system. This virtual framework is a crucial step in providing DMC as an experimentation platform for distributed quantum computations, as reasoning within the formal model by hand proves quite cumbersome for even small computations. At the basis of our work lies the measurement calculus (Danos *et al.*, 2007), a low-level quantum programming language for measurement-based quantum computations from which we explore the distributed paradigm. Next to the obvious advantage of starting from a proper formal framework, measurement-based models are well-suited as a starting point for distributed quantum computations because they are inherently distributed. What is important here is that the well-known physical framework of quantum computation is ported to an equivalent computer science framework, opening up the field towards investigations from this branch of science as well.

The structure of this paper is as follows. In the next section we discuss the syntax of our language, while Sec. 3 covers the semantics of DMC and lists its properties. Sec. 4 applies the previous to a concrete example, a composite protocol implementing distributed controlled gates. We discuss the implementation of the quantum virtual machine for DMC in Sec. 5, and conclude in Sec. 6. Some basic knowledge of quantum computation is assumed; for the reader not familiar with the domain we refer to the excellent (Nielsen and Chuang, 2000). Our approach in this article is to explain the notions of the model by example, rather than providing a series of formal definitions which are hard to interpret and for which space it too limited. However, we stress that the model is a rigorous one, and, while this paper is a stand-alone document, refer the interested reader to the appendix for the full formal semantics of the DMC language and (D'Hondt, 2005; Danos *et al.*, 2005) for complete definitions.

2 Syntax

The language we propose is, broadly speaking, an assembly language for distributed measurement-based quantum computations. It is an assembly language in that it provides syntax only for the most basic operations while at the same time being universal. It is measurement-based as we build our language on top of the measurement calculus (MC) (Danos *et al.*, 2007), an assembly language for measurement-based quantum computations. The latter depart from the usual circuit-based approach to quantum computing, where unitary transformations are the driving force of computations. While measurement operations were long seen as a necessary but disruptive part of quantum computing, in algorithms such as teleportation they act as an essential part of the computation. This gave rise to models where the computation is steered by pre-established generic entanglement combined with measurements, such as the one-way quantum computer (Raussendorf *et al.*, 2003). Because measurements are inherently probabilistic, correction operations are required that are conditionally applied depending on previous measurement outcomes, thus rendering the computation deterministic. All of this is nicely captured in the measurement calculus. For this reason, as well as the inherent distributive aspect of measurement-based models, MC is an ideal basis from which to develop our language.

We describe our language model in three layers. The base layer consists of MC *patterns*, which describe local quantum computations. These are combined with distribution primitives into the notion of *agents* in the middle layer. Finally, agents and their shared entanglement resources are bundled into *networks* in the top layer.

Patterns. In MC a pattern is defined by a sequence of commands together with sets of qubits for working, input and output memory. As an example consider the following pattern, which, as we explain below, implements the Hadamard operation,

$$\mathcal{H}(1,2) := (\{1,2\},\{1\},\{2\},X_2^{s_1}M_1^X E_{12}) \ . \tag{1}$$

All qubits are uniquely identified using numbers. The first argument denotes that the pattern has a computation space of two qubits, referenced by 1 and 2. The next two arguments specify the pattern's inputs and outputs. Working qubits that are not input qubits are initialised to $|+\rangle = |0\rangle + |1\rangle$. The last argument is the pattern's command sequence, a list of operations taken from a basic set and executed from right to left, analogous to matrix applications. Subscripts denote qubit arguments of the operation, while corrections are conditionally executed depending on their superscript. Concretely, one runs the pattern $\mathcal{H}$ by preparing the first qubit in some input state $|\psi\rangle$ and the second in state $|+\rangle$, then entangling both qubits with the controlled-Z operation[1] to obtain $CZ_{12}(|\psi\rangle \otimes |+\rangle)$ (syntax: E_{12}). Next, the first qubit is measured in the $\{|+\rangle, |-\rangle\}$ basis (syntax: M_1^X), where $|-\rangle = |0\rangle - |1\rangle$. Finally, an X correction is applied on the output qubit if the measurement outcome was s_1 (syntax: $X_2^{s_1}$). Here s_1 defines a *signal* – simply 0 or 1 – coming from the X-basis measurement on qubit 1 in M_1^X. If the signal is 0, the operation is not performed. Measurements are considered to be destructive, which is why qubit 1 does not appear in the output set.

A general pattern is denoted $\mathcal{P}(V, I, O, \mathcal{A})$, with computation space V, inputs I and outputs O (together called the *pattern type*), and command sequence $\mathcal{A}$ that consists of *entanglements* E_{ij}, *measurements* ${}^t[M_i^\alpha]^s$, or *corrections* X_i^s or Z_i^s, where $i, j \in V$, $\alpha \in [0, 2\pi]$ and $s, t \in \mathbb{Z}_2$. Allowed measurements are those in the XY-plane of the Bloch sphere, and are specified by the angle α (M_1^X is actually syntactic sugar for M_1^0). Measurement angles may also be conditioned by signals, written ${}^t[M_i^\alpha]^s$, with $(-1)^s\alpha + t\pi$ being the actual measurement angle. The four basic instructions together with signal conditioning suffice to make the model universal (Raussendorf *et al.*, 2003; Danos *et al.*, 2007).

Patterns can be combined into larger ones to create arbitrary quantum computations. The *sequential composition* of patterns $\mathcal{P}_1 = (V_1, I_1, O_1, \mathcal{A}_1)$ and $\mathcal{P}_2 = (V_2, I_2, O_2, \mathcal{A}_2)$, with $O_1 = I_2$, is defined as

$$\mathcal{P}_2\mathcal{P}_1 := (V_1 \cup V_2, I_1, O_2, \mathcal{A}_2\mathcal{A}_1) \ , \tag{2}$$

while the *parallel composition* of the same patterns is defined as

$$\mathcal{P}_1 \otimes \mathcal{P}_2 := (V_1 \uplus V_2, I_1 \uplus I_2, O_1 \uplus O_2, \mathcal{A}_1\mathcal{A}_2) \ . \tag{3}$$

Note that one can always rename qubits for parallel composition to become possible, while sequential composition also needs I_2 and O_1 to have the same cardinality.

As an example, here is the pattern to create a 3-fold GHZ-state $|000\rangle + |111\rangle$,

$$GHZ_{123} = (\{1, 2, \hat{2}, 3, \hat{3}\}, \cdot, \{1, 2, 3\}, \mathcal{H}(\hat{3}, 3)E_{2\hat{3}}\mathcal{H}(\hat{2}, 2)E_{1\hat{2}}) \ , \tag{4}$$

where $\hat{2}$ and $\hat{3}$ are working qubits.

MC is equipped with a powerful standardisation theorem which provides a procedure for bringing any pattern, such as the one above, into EMC-form, i.e. with entanglements first and corrections last. This is important from an experimental implementation perspective, and also corresponds nicely with the typical structure of a distributed quantum protocol where one starts out with a shared entanglement resource. In fact within MC we can already express distributed computations such as teleportation,

$$(\{1, 2, 3\}, \{1\}, \{3\}, X_3^{s_2}Z_3^{s_1}M_2^X M_1^X E_{12}E_{23}) \ . \tag{5}$$

While the pattern describes the intended computation, we find no notion of separate parties participating in the teleportation, and neither of the required communication between them. Purely by convention, and because everybody knows the protocol, we can say that qubits 1 and 2 belong to Alice and qubit

[1]A controlled-Z operation on two qubits applies the Z operation to the second qubit provided the first is set to 1.

3 to Bob. It is still hard to see if and what both parties have to communicate, e.g. Bob needs s_1 and s_2 but has no access to qubits 1 and 2, thus Alice needs to perform the measurements and communicate the results to Bob. We want this information to be explicit and obvious in the language, which in turn makes it easier to describe and reason about distributed quantum programs. To see that this is important, try to identify what is implemented with the following distributed protocol,

$$(\{0,\bar{0},1,\bar{1},2,\bar{2}\},\cdot,\{0,1,2\},Z_2^{s_{\bar{2}}}Z_1^{s_{\bar{1}}}Z_0^{s_{\bar{0}}}M_{\bar{0}\bar{1}\bar{2}}^{GHZ}E_{0\bar{0}}E_{1\bar{1}}E_{2\bar{2}})\ . \tag{6}$$

Here M^{GHZ} is a pattern for GHZ-measurement. Note that pattern (4) implements the unitary transformation between the diagonal basis and the GHZ-basis $\{|i\rangle+|\bar{i}\rangle\}_{i=0}^{2^n-1}$. Hence a GHZ-measurement is executed by applying the inverse pattern followed by a measurement in the diagonal basis. We will get back to this pattern in Sec. 4.

Agents. An agent embodies a single computation node running in isolation in a distributed algorithm, i.e. a processor or a party such as Alice or Bob. Each agent has a local command sequence, which operates on a set of resources contained within its environment. Agent instructions are either measurement patterns or communication primitives. For example, the teleportation pattern (5) really needs the following agent definitions for Alice and Bob, respectively,

$$\mathbf{A}:\{1,2\}.\mathsf{c}!s_2.\mathsf{c}!s_1.M_2^X M_1^X E_{12} \quad \text{and} \quad \mathbf{B}:\{3\}.X_3^{s_2}Z_3^{s_1}.\mathsf{c}?s_2.\mathsf{c}?s_1\ . \tag{7}$$

As we can see an agent definition consists of a *type* in curly braces, which specifies what qubits an agent owns, and an instruction sequence. Next to local quantum operations, written in MC language, we specify the exchange of the signals s_1 and s_2 between Alice and Bob via the communication primitives $\mathsf{c}?$ and $\mathsf{c}!$. What is missing here as compared to pattern (5) is the prior shared entanglement E_{23}, as indeed it is impossible to factorise this part of the protocol into agent definitions. There is no other option than to specify shared entanglement separately in the network definition for the full protocol, as we shall see below. Of course the qubits are still local to the agents – as reflected in the type – only their description is not.

Formally, an agent is defined by an expression $\mathbf{A}(i,o):Q.\mathcal{E}$, where the type Q is a set of qubit references and $\mathcal{E}$ is a finite instruction sequence composed of pattern command sequences $\mathcal{A}$, classical message reception $\mathsf{c}?s$ and sending $\mathsf{c}!s$, where s is a signal, and quantum reception $\mathsf{qc}?q$ and sending $\mathsf{qc}!q$, where q is a qubit reference. Corresponding communication actions are synchronised, meaning that an agent executing a receive will pause its program until the required agent has sent a message on the same channel and vice versa. Classical inputs i and outputs o are used in protocols such as superdense coding, and are not mentioned when there are none. Also, working qubits required by patterns are initialised to $|+\rangle$ as before and are not specified in the type Q.

Networks. A network of agents consists of several agents that execute their command sequence concurrently. Typically, quantum resources are shared between agents – indeed most distributed quantum protocols benefit from some type of shared entanglement being present prior to the start of the protocol. We have no way of splitting the representation of these states over agent definitions, and instead have to specify agents separately in a network's definition. This is why a network specification is more than just a collection of agent definitions, a feature specific to quantum concurrent frameworks and absent in classical concurrency. To keep track of how a shared resource is distributed each agent's type Q specifies which qubits of a shared resource are local to that agent.

At this point we can finally write down the full teleportation protocol, including all relevant distributed aspects.

$$\begin{aligned} TP := \ & \mathbf{A} : \{1,2\}.\mathsf{c}!s_2.\mathsf{c}!s_1.M_2^X M_1^X E_{12} \\ & \mid \mathbf{B} : \{3\}.X_3^{s_2} Z_3^{s_1}.\mathsf{c}?s_2.\mathsf{c}?s_1 \\ & \parallel E_{23} \end{aligned} \tag{8}$$

Analogous to process algebraic notation we use a vertical bar | to separate concurrently executing agents. Shared network resources are specified after the double bar ‖. For teleportation this is the state E_{23}, which was produced and handed out to Alice and Bob prior to the execution of the protocol. If so desired the establishment of shared resources can itself be seen as a distributed protocol involving a server agent polled for entanglement services. We now likewise lift the pattern given in (6) to a distributed setting, obtaining the following

$$\begin{aligned} ES := \ & \mathbf{L} : \{\bar{0},\bar{1},\bar{2}\}.\mathsf{c}_0!s_{\bar{0}}.\mathsf{c}_1!s_{\bar{1}}.\mathsf{c}_2!s_{\bar{2}}.M_{\bar{0}\bar{1}\bar{2}}^{GHZ} \ . \\ & \mid \mathbf{A}_0 : \{0\}.Z_0^{x_{\bar{0}}}.\mathsf{c}_0?x_{\bar{0}} \\ & \mid \mathbf{A}_1 : \{1\}.Z_1^{x_{\bar{1}}}.\mathsf{c}_1?x_{\bar{1}} \\ & \mid \mathbf{A}_2 : \{2\}.Z_2^{x_{\bar{2}}}.\mathsf{c}_2?x_{\bar{2}} \\ & \parallel E_{0\bar{0}} E_{1\bar{1}} E_{2\bar{2}} \ , \end{aligned} \tag{9}$$

where $M_{\bar{0}\bar{1}\bar{2}}^{GHZ}$ is a pattern executing a GHZ-measurement. This is the entanglement swapping protocol for three agents (Zukowski *et al.*, 1993; Bose *et al.*, 1998), which produces a GHZ-state shared between $\mathbf{A}_0$, $\mathbf{A}_1$ and $\mathbf{A}_2$. Because the resulting GHZ state is in the diagonal basis we have Z rather than X corrections in the network specification.

Arbitrary networks are denoted $\mathcal{N} = |_i \mathbf{A}_i(\mathbf{i}_i, \mathbf{o}_i) : Q_i.\mathcal{E}_i \parallel \sigma$. For simplicity, we assume that networks of agents satisfy a number of definiteness conditions which ensure that the computation is well-defined. For example, an agent may only operate on qubits he owns, and every communication event needs to have a corresponding dual event in the network. Of course these issues are important but we are glossing over them here as we assume checking these occurs in a pre-compilation step rather than at runtime.

Our extensions to the measurement calculus have made distributed notions explicit. We can now see directly from a protocol specification what agents have to communicate to whom as well as which assumptions are made about non-local entanglement resources. Also, because each agent's instructions are expressed separately, we do not impose any particular execution order for local quantum operations. This makes it clear which parts of the protocols can and cannot run concurrently.

3 Semantics

In the previous section we established the syntax of DMC. The next step is to establish the meaning of a program written down in this language, i.e. its semantics. Formal semantics is a means of assigning objects to chunks of code so that one can reason with these objects rather than with the code itself. In this section we describe an operational as well as a denotational semantics for DMC programs and, with this in hand, state some important properties of our framework.

Definition. A network's operational semantics reflects how it affects the state of a distributed system on which the network is run. That is, we associate with each agent a local state (quantum and classical), and specify how the network specification updates these. Since we always have to take the shared entanglement resources into account the operational semantics does not decompose into state transformers for each of the agents separately. On top of this a crucial ingredient of the semantics specifies how quantum resources move throughout the system. Concurrency is not really an issue for the semantics because, due to the linearity of quantum mechanics, the order in which local operations are applied is unimportant. Hence we pick an order arbitrarily and derive the semantics for this case.

Concretely, the operational semantics of a network is found by collapsing individual small-step semantical rules into one transition system. The full small-step semantics of DMC can be found in the appendix; it consists of a number of quantum mechanical rules – corresponding to measurement pattern commands – and a number of communication rules. These affect the global quantum state (EMC commands), the local resources (quantum communication), and the local memory (M commands and classical communication). Since measurement is probabilistic, so are the small-step rules. The formal semantics of measurement pattern commands is well established (Danos *et al.*, 2007; D'Hondt, 2005), and just needs to be lifted to the DMC setting. We elaborate here the semantics of typically distributed concepts introduced in the previous section. The formal semantics of MC relies on the component Γ, the outcome map, which contains a number of bindings from signal names to measurement outcomes. In DMC the outcome map is lifted to a local agent memory recording also the values of classical messages. Concretely, a classical communication event between two agents has the following semantics, given that Γ_b, the local memory of Bob, evaluates the name y to the value v,

$$\begin{aligned}&\mathbf{A} : (Q_a, \Gamma_a).\mathcal{E}_a.\mathsf{c}?x \mid \mathbf{B} : (Q_b, \Gamma_b).\mathcal{E}_b.\mathsf{c}!y \\ \Longrightarrow\ &\mathbf{A} : (Q_a, \Gamma_a[x \to v]).\mathcal{E}_a \mid \mathbf{B} : (Q_b, \Gamma_b).\mathcal{E}_b \quad .\end{aligned} \tag{10}$$

Here $\Gamma_a[x \downarrow v]$ means that we assign a new binding of x to v in the local memory of Alice. These rules are typically much harder to read and write down formally than to apply concretely. What we are saying here is that if a classical communication takes place, the value is looked up by the sending agent, and received and bound to a new name by the receiving agent, after which the computation ($\mathcal{E}_a$ and $\mathcal{E}_b$) continues. Sending and receiving qubits over a quantum channel changes the types Q of the agents involved. An agent sending a qubit can no longer perform any operations on it, therefore the corresponding qubit reference is removed from Q, while it is added to the receiving agent. Formally, the rule for the exchange of a qubit with reference i is given by

$$\begin{aligned} &\mathbf{A} : (Q_a, \Gamma_a).\mathcal{E}_a.\mathsf{qc}?i \mid \mathbf{B} : (Q_b, \Gamma_b).\mathcal{E}_b.\mathsf{qc}!i \\ \Longrightarrow\ &\mathbf{A} : (Q_a \cup \{i\}, \Gamma_a).\mathcal{E}_a \mid \mathbf{B} : (Q_b \backslash \{i\}, \Gamma_b).\mathcal{E}_b \ \ . \end{aligned} \tag{11}$$

It goes without saying that the way in which the actual communication of a qubit is implemented is more intricate than this simple rule. However, by providing quantum communication as primitive syntax we are precisely choosing not to get into these implementation matters. Both rules mentioned here do not affect the global quantum state, which is why it is not mentioned. This is not the case for the small-step semantics for pattern commands, which moreover may be probabilistic. The full small-step semantics can be found in the appendix or in (D'Hondt, 2005; Danos *et al.*, 2005).

We obtain the operational semantics by defining computation paths in the usual way and gathering all small steps in a computation path in one big step from initial to final state in that path. As such the operational semantics of a quantum distributed network is essentially a probabilistic transition system (PTS). However, since resource allocation is crucial, we have to augment this PTS with information on how qubits move throughout the network. This is formalised as a type signature. We denote the operational semantics of a network $\mathcal{N}$ by $[\![\mathcal{N}]\!]_{op}$. For example, the operational semantics of TP is given by the deterministic transition system

$$[\![TP]\!]_{op} : (\{1\}, \cdot) \to (\cdot, \{3\}) \ . \ \rho_1 \Longrightarrow \rho_3 \ , \tag{12}$$

where ρ is the density matrix specifying the input quantum state to be teleported and subscripts indicate qubit systems. The operational semantics of the ES network given in (9) is also deterministic.

$$[\![ES]\!]_{op} : (\cdot, \cdot, \cdot, \cdot) \to (\cdot, \{0\}, \{1\}, \{2\}) \ . \ \mathbf{0} \Longrightarrow GHZ^D_{012} \ , \tag{13}$$

where the superscript D denotes that the resulting state is in the diagonal basis. Note that we only write real quantum inputs in the type while not mentioning entanglement resources. The good thing about operational semantics is that in principle it can be derived automatically by induction on the small-step rules.

Denotational semantics is a second means of assigning a formal meaning to a chunk of code, this time by way of mathematical objects. If we look at the skeleton of a distributed protocol, i.e. the equivalent non-distributed pattern, we find a multi-local probabilistic quantum operation, which is mathematically represented by (a special type of) completely positive map (CPM). Since protocols with different distribution of resources are observationally different, we have to pair this CPM with a function mapping input to output resources, formally represented by a type signature. We denote this type of semantics by $[\![\mathcal{N}]\!]_{de}$ for a network $\mathcal{N}$. For example, the denotational semantics of TP is given by the map

$$[\![TP]\!]_{de} : (\{1\}, \cdot) \to (\cdot, \{3\}) \ . \ \mathcal{I} \ , \tag{14}$$

i.e. TP implements the identity map from qubit 1 to qubit 3. Here we see the importance of the type signature in specifying the semantics. The subtle difference between both types of semantics becomes more apparent once one starts investigating more complex protocols involving mixed states and probabilistic semantics.

As we see from the examples both types of semantics are quite similar, and indeed we have the following result, which we state without proof.

Proposition 3.1 *There is a* precise correspondence *between the operational and the denotational semantics of networks of agents, that is to say*

$$\forall \mathcal{N}_1, \mathcal{N}_2 : \mathcal{N}_1 \equiv_{op} \mathcal{N}_2 \iff \mathcal{N}_1 \equiv_{de} \mathcal{N}_2 \ . \tag{15}$$

Two networks are semantically equivalent if they have the same semantics.[2] While the equivalence between semantics may seem obvious (certainly from the example), it is crucial to prove this statement formally, as it allows one to switch between semantics where appropriate without giving the issue further thought. We note that in general this equivalence is not guaranteed.

Building larger protocols. With syntax and semantics in hand we can now consider constructing larger networks from smaller components. We consider parallel as well as sequential composition of networks, denoted by $\otimes$ and $\circ$ respectively. These operations are defined in the obvious way so we will not spell out concrete definitions here. Suffice to say that one needs to pay attention to agent as well as qubit names to ensure that networks are connected as desired. More importantly we need to make sure that these forms of composition are consistent with the semantics, as is indeed the case.

Proposition 3.2 *The semantics of networks is* compositional*, i.e.*

$$[\![\mathcal{N}_2 \circ \mathcal{N}_1]\!] = [\![\mathcal{N}_2]\!] \circ [\![\mathcal{N}_1]\!] \ . \tag{16}$$

$$[\![\mathcal{N}_1 \otimes \mathcal{N}_2]\!] = [\![\mathcal{N}_2]\!] \otimes [\![\mathcal{N}_1]\!] \tag{17}$$

Here we have a first application of Prop. 3.1, as first we do not need to specify which type of semantics we mean in the statement of Prop. 3.2 , and second we can choose the most convenient semantics for the proof, which is the denotational semantics. While the compositions on the left hand side are at the level of agent and network definitions, the composition of network semantics on the right hand side is a functional one at the level of type signatures, PTS's and CPMs. For the full proof we refer to (D'Hondt, 2005); an example is given in Sec. 4.

A second important notion when constructing larger protocols, closely related to compositionality, is that of *contexts*. Informally speaking a context is a program with a "hole" in which a network specification can be inserted, typically denoted $C[\cdot]$. On some occasions, one finds that programs that are considered equivalent in isolation no longer behave in the same way when placed within the context of a larger program. This is particularly the case in concurrent systems. A historical example is the *Brock-Ackerman anomaly* (Brock and

[2]For example, one can prove that qubit communication between two agents is semantically equivalent to the teleportation network.

Ackerman, 1981), by which it was realised that simple input-output behaviour is not enough to reason about equivalence of concurrent systems. In our framework contexts $C[\mathcal{N}]$ are given by arbitrary network compositions C in which our network $\mathcal{N}$ is placed. What is different with ordinary compositionality is that the quantum resources of the network $\mathcal{N}$ are no longer considered to be independent of those of its context. Indeed in Prop. 3.2 quantum resources are considered to be provided independently, such that composite networks operate on disentangled inputs. While this is sensible when each of the networks operate in isolation, it is less so when a network is only one factor in a complex compositional structure. Hence we first need to show that our semantics is independent of so-called *entanglement contexts*. This is a consequence of the following proposition, which we state here without proof.

Proposition 3.3 *Suppose* $\mathcal{L} : \rho_A \longrightarrow \rho_B = \sum_k L_k \rho_A L_k^\dagger$ *is a completely positive map. Then for all quantum states* ρ_{AC} *applying* $\mathcal{L}$ *to the A-part of* ρ_{AC} *results in*

$$\rho_{BC} = \sum_k (L_k \otimes I_C)\rho_{AC}(L_k^\dagger \otimes I_C). \tag{18}$$

The proof, though easy, is not trivial. Using this proposition and compositionality we have the following important result.

Corollary 3.4 *Equivalence holds in arbitrary contexts, that is, if* $\mathcal{N}_1 \equiv \mathcal{N}_2$, *then for all network contexts* $C[\cdot]$ *we have that* $C[\mathcal{N}_1] \equiv C[\mathcal{N}_2]$.

4 Applications

We now show how all of the formal ingredients can be put to use in a concrete example. The distributed primitive we will investigate is that of a *distributed remotely controlled gate* (Yimsiriwattana and Lomonaco, 2005). Near-future quantum computers are expected to have only a limited number of qubits per machine. Even in quantum simulating environments the current qubit limit is only about 36 (Raedt *et al.*, 2006). Hence one can imagine that quantum computations need to proceed much like cluster computations today, with resources spread over different processors in order to make them feasible. One common situation would be where the central processor needs to execute a controlled operation with the target qubits spread over a group of agents. Following the ideas in (Yimsiriwattana and Lomonaco, 2005), once we have a GHZ resource we can execute a distributed controlled gate. Calling the central processor **L** (for Leader), and assuming there are two subordinate processors **A** and **B**, the trick is to establish a shared control qubit between target agents, which is achieved through the *share control* (SC) protocol as follows,

$$\begin{aligned} SC := \;\; & \mathbf{L} : \{c, 0\}.\mathtt{c}_{1,2}!s_0.\, M_0^X E_{0c}\; . \\ & \mid \mathbf{A}_1 : \{1\}.X_1^{x_0}.\mathtt{c}_1?x_0 \\ & \mid \mathbf{A}_2 : \{2\}.X_2^{x_0}.\mathtt{c}_2?x_0 \\ & \parallel GHZ_{012}^D \end{aligned} \tag{19}$$

Here qubit c is the input control qubit which is in the state $\alpha|0\rangle + \beta|1\rangle$. That the protocol indeed establishes its goal may be seen from its semantics, which can be derived unambiguously to be

$$\begin{aligned}\llbracket SC\rrbracket_{op} : (\{c\},\cdot,\cdot) &\to (\{c\},\{1\},\{2\}) \\ (\alpha|0\rangle + \beta|1\rangle)_c &\Longrightarrow (\alpha|000\rangle + \beta|111\rangle)_{c12} \ .\end{aligned} \tag{20}$$

Once this type of shared entanglement is in place each target agent just has to execute a local controlled unitary gate with as control its qubit in the shared entanglement resource. Note that because we have context-independence of the semantics, target qubits may be entangled over different agents. However, in order for the distribution approach to be possible at all, the controlled unitary CU must have $U = U_1 \otimes U_2$ where, U_1 and U_2 operate on a number of target qubits smaller or equal than the maximum available qubits for each agent.

We see that the SC protocol requires GHZ entanglement. It is probably realistic to assume that in a quantum network each agent can ask for Bell-state entanglement with the central server. It is not so realistic to assume that groups of agents can demand direct GHZ entanglement whenever they need it; rather, we expect GHZ entanglement to be produced via the entanglement swapping protocol. Note that, since we need GHZ entanglement between $\mathbf{L}$, $\mathbf{A}_1$ and $\mathbf{A}_2$, we need to compose $\mathbf{L}$ with $\mathbf{A}_0$ in the ES protocol as presented earlier in (9), that is

$$\mathbf{L} := \mathbf{A}_0 \circ \mathbf{L} : \{0,\bar{0},\bar{1},\bar{2}\}.Z_0^{x_{\bar{0}}}.\mathsf{c}_1!s_{\bar{1}}.\mathsf{c}_2!s_{\bar{2}}.M^{GHZ}_{0\bar{1}\bar{2}} \ . \tag{21}$$

What we are actually doing here is composing the SC protocol with no shared resource with the ES protocol to establish the resource, and our semantics ensures that this is something we can do unambiguously. Indeed, we have

$$\begin{aligned}&\llbracket SC \circ ES\rrbracket = \llbracket SC\rrbracket \circ \llbracket ES\rrbracket \\ &= (\{c\},\cdot,\cdot) \to (\{c\},\{1\},\{2\}) \\ &\quad . \ (\alpha|0\rangle + \beta|1\rangle)_c \Longrightarrow (\alpha|000\rangle + \beta|111\rangle)_{c12},\end{aligned} \tag{22}$$

with $\llbracket SC\rrbracket$ and $\llbracket ES\rrbracket$ given in (20) and (13) respectively.

More agents. The networks that we defined for implementing the distributed remote gate protocol can be generalised to n agents. This requires generalised procedures for GHZ-measurement, entanglement swapping and establishing a shared control qubit. First, an n-fold GHZ state is produced by generalising the 3-GHZ-pattern given in (4). That is, through the pattern with no input qubits, output qubits $\{1, 2, \ldots, n\}$ and an event sequence interleaving E and H operations, as follows,

$$GHZ_{1\ldots n} = \mathcal{H}(\hat{n}, n)E_{(n-1)\hat{n}} \ldots E_{3\hat{4}}\mathcal{H}(\hat{3}, 3)E_{2\hat{3}}\mathcal{H}(\hat{2}, 2)E_{1\hat{2}} \ , \tag{23}$$

where the hatted qubits are again working qubits. Again, a GHZ-measurement is executed by applying the inverse pattern followed by a diagonal-basis measurement. This leads to the pattern

$$M^{GHZ}_{1\ldots n} = M^0_{\hat{1}}M^0_{\hat{2}} \ldots M^0_{\hat{n}}E_{1\hat{2}}\mathcal{H}(2,\hat{2})E_{2\hat{3}}\mathcal{H}(3,\hat{3})E_{3\hat{4}} \ldots E_{(n-1)\hat{n}}\mathcal{H}(n,\hat{n}) \ , \tag{24}$$

with input qubits the qubits $\{1, 2, \ldots, n\}$ to be measured and with no outcome qubits. Using this sub-pattern we can establish GHZ-entanglement between the leader $\mathbf{L}$ and agents $\mathbf{A}_1$ through to $\mathbf{A}_n$ by the generalised entanglement swapping protocol (Zukowski *et al.*, 1993; Bose *et al.*, 1998), which has the following network specification,

$$\begin{aligned} ES := \quad & \mathbf{L} : \{0, \bar{0}, \ldots, \bar{n}\}.Z_0^{x_{\bar{0}}}.(\mathtt{c_i}!s_{\bar{i}})_{i=1}^{n}.M_{\bar{0}\ldots\bar{n}}^{GHZ} \\ & |_{i=1}^{n} \; \mathbf{A}_i : \{i\}.Z_i^{x_{\bar{i}}}.\mathtt{c_i}?x_{\bar{i}} \\ & \| \;\; \otimes_{i=0}^{n} E_{i\bar{i}} \; . \end{aligned} \tag{25}$$

This is just the generalisation of network (9) where we have merged agents $\mathbf{L}$ and $\mathbf{A}_0$ for the purpose of establishing a shared control qubit as before. The signal $s = s_{\bar{0}} \ldots s_{\bar{n}}$ corresponds to a projection on the GHZ-state $|s\rangle + |\overline{s}\rangle$. This network has the following semantics,

$$[\![ES]\!]_{op} : (\cdot, \ldots, \cdot) \to (\{0\}, \{1\}, \ldots, \{n\}) \; . \; \mathbf{0} \Longrightarrow GHZ_{0\ldots n}^{D} \; . \tag{26}$$

After establishing GHZ-entanglement between agents a shared control qubit is obtained through the following protocol.

$$\begin{aligned} SC := \quad & \mathbf{L} : \{c, 0\}.\mathtt{c_i}!s_0.\, M_0^X E_{0c} \; . \\ & |_{i=1}^{n} \; \mathbf{A}_i : \{i\}.X_i^{x_0}.\mathtt{c_i}?x_0 \\ & \| \; GHZ_{0\ldots n}^{D} \end{aligned} \tag{27}$$

The semantics of this network is given by

$$\begin{aligned} & [\![SC]\!]_{op} : (\{c\}, \cdot, \ldots, \cdot) \to (\{c\}, \{1\}, \ldots, \{n\}) \\ & \qquad (\alpha|0\rangle + \beta|1\rangle)_c \Longrightarrow (\alpha|0\rangle^{\otimes(n+1)} + \beta|1\rangle^{\otimes(n+1)})_{c1\ldots n} \; . \end{aligned} \tag{28}$$

Control qubit c can now indirectly control unitary operations at the sites of all agents by having agent $\mathbf{A}_i$ execute a local pattern for a controlled unitary gate where the control is its qubit i and the targets are locally available qubits. Again, for this approach to be possible the controlled unitary CU must have $U = U_1 \otimes \ldots \otimes U_n$ where each of the sub-unitary U_i is executable by agent $\mathbf{A}_i$.

5 Virtualisation

In the above we introduced a formal language for distributed quantum computation. Providing a number of tools for constructing higher-level programs, it should be seen as a first step in a bottom-up construction of a distributed quantum programming paradigm. However, the previous section clearly shows how cumbersome it is to describe and evaluate computations purely within the formal model. Rather, these developments are only really useful when one thinks of the language DMC in terms of a quantum virtual machine (QVM). A *virtual machine* is a platform-independent programming environment that abstracts away details of the underlying hardware or operating system. In our setting it is a low-level language abstraction layer which executes DMC programs independent of the actual implementation of quantum operations, which could be executed by any of several existing quantum simulators or even by a physical quantum computer. A QVM forms a crucial layer in a tiered quantum computation architecture (Svore *et al.*, 2006) by acting as mediator between a set of low-level basic quantum gates and the construction of more complex quantum programs,. This is invaluable if one is concerned with developing higher-level distributed quantum computation languages through experimentation and abstraction. Using the QVM as a firm basis we show we can build abstraction layer upon layer, towards an invaluable user-friendly and flexible programming environment.

Translating the formal model into a virtual machine has the obvious benefits of automating program execution and composition. Furthermore, low-level inspection during a pre-compilation step can automatically determine a number of issues such as well-definedness of code. While the formal language we have discussed in the above provides the backbone for our virtual machine, its actual implementation is far from trivial. Issues such as well-definedness of programs, identifiers and variables within local computations as well as in larger program contexts and efficiency all come into play in a more concrete sense that is absent in the more abstract formal model. We will dwell on such elements of our implementation where it essentially and significantly differs from the formal DMC model. Note that we use the adjectives *formal* and *virtual* to differentiate between similar objects in the formal model and the virtual one whenever the context is unclear.

Instead of developing an ad-hoc computer program to execute the formal DMC language directly, we have opted for a layered design that clearly separates each feature of the language into a separate 'compilation' step from a more abstract version of the language into a more concrete one. We use this to our advantage; first implementing a stand-alone MC virtual framework and then formulating the DMC's as an extension, as we have done for the formal model. The structure of the rest of this section reflects this design, dealing first with an execution then a composition layer for sequential MC-based computations in Sec. 5.2. Next we extend this platform towards a distributed version thereof in Sec. 5.3. In Sec. 5.4 we expose the implementation specific details and specify the syntax and semantics of each layer in turn. In the final part of this section we show a user-friendly tool to graphically design computations, based on our platform. However, we will first give an overview of our virtual framework's design.

5.1 Overview

Setting aside our graphical toolkit shown in Sec. 5.5, our virtual framework can be seen as a compiler suite; compiling a more abstract and expressive language into a small assembly-like execution language. This execution language can be seen as the instruction set architecture of a low-level Quantum Virtual Machine, forming the basis layer in our design. The design of the full DMC virtual environment is presented here incrementally, starting with the virtualisation of the MC and then moving on to the distribution extensions forming the DMC virtualisation.

We first focus on the execution layer for the MC. To design this we have to turn the formal MC language into a more concrete form. Taking as a running example the Hadamard pattern from Eq.(1)

$$\mathcal{H}(1,2) := (\{1,2\},\{1\},\{2\},X_2^{s_1}M_1^X E_{12}) ,$$

becomes the computer readable expression

```
((E 1 2) (M 1 0) (X 2 (s 1))) .
```

The exact syntax and semantics of this and following examples are elaborated on in Sec. 5.2.

Measurement pattern expressions and the composition thereof are added in a separate layer, as the pattern abstraction is not needed during execution. The measurement pattern abstraction introduces qubit sorts and local qubit references. The Hadamard command sequence example above is expressed as a pattern

```
((?i ?o) (?i) (?o) ((E ?o ?i) (M ?o 0) (X ?o (s ?i)))) ,
```

which can be easily compiled into the above executable expression by taking only the command sequence and replacing the variable names, preceded by a question mark above, by concrete and unique qubit identifiers. This is performed by the composition layer on which we focus ourselves in Sec. 5.2.2. Measurement patterns are introduced as a way to compose multiple command sequences effectively, by merging several patterns into a singleton. For example, the Controlled-NOT pattern can be expressed as a combination of the Hadamard and a Controlled Z operation

$$\wedge\mathcal{X} := (\mathcal{I}(1) \otimes \mathcal{H}(3,4)) \circ \wedge\mathcal{Z}(1,3) \circ (\mathcal{I}(1) \otimes \mathcal{H}(2,3)) \ . \tag{29}$$

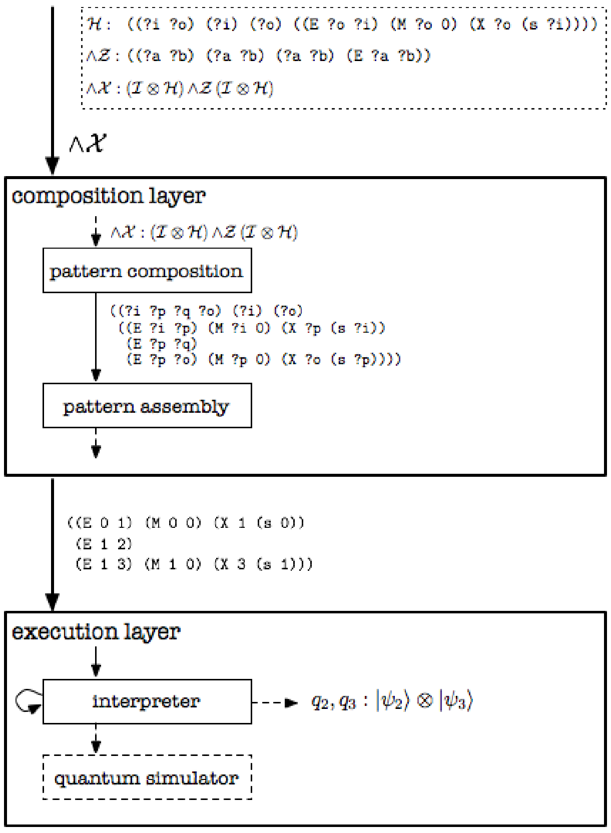
The composition mechanics are explained further in Sec. 5.2.2. Combining the execution and composition layer we can illustrate the design for the virtual MC framework in Figure 1.

For the DMC virtual framework in Sec. 5.3 we expand the stand-alone MC virtual framework above horizontally. That is, extending each layer instead of adding new ones. The new executable language for the execution layer now contains multiple command sequences that need to be executed concurrently, instead of the one command sequence. Each command sequence expresses the computation performed by a different agent in the network. The set of operations in the execution layer's language is extended with the basic communication primitives. The pattern layer is extended with support for agent patterns and network composition. The former adds channels next to qubit variables, the latter expresses how qubits and channels are shared across agents in the network. Similar to the MC virtual framework, a network is composed and compiled down to the underlying execution language. Agents as they appear in formal DMC become a type of pattern that can not only be composed by their in- and output qubits, but also by communication channels. Agents are composed with each other and with the global resource pattern $\mathcal{R}$, according to the network configuration NC. Figure 2 illustrates this with the classic teleportation example.

The definition of pattern compositions lend themselves well to a graph-based visual notation, as we will see. We exploit this fact with a Graphical User Interface(GUI) in Sec. 5.5, avoiding a cluttered textual notation.

5.2 The quantum virtual machine

We first present a virtual environment for the MC, providing a way to compose measurement patterns and execute the result. This environment implements



```
H:  ((?i ?o) (?i) (?o) ((E ?o ?i) (M ?o 0) (X ?o (s ?i))))
∧Z: ((?a ?b) (?a ?b) (?a ?b) (E ?a ?b))
∧X: (I ⊗ H) ∧Z (I ⊗ H)
```

```
((?i ?p ?q ?o) (?i) (?o)
 ((E ?i ?p) (M ?i 0) (X ?p (s ?i))
  (E ?p ?q)
  (E ?p ?o) (M ?p 0) (X ?o (s ?p))))
```

```
((E 0 1) (M 0 0) (X 1 (s 0))
 (E 1 2)
 (E 1 3) (M 1 0) (X 3 (s 1)))
```

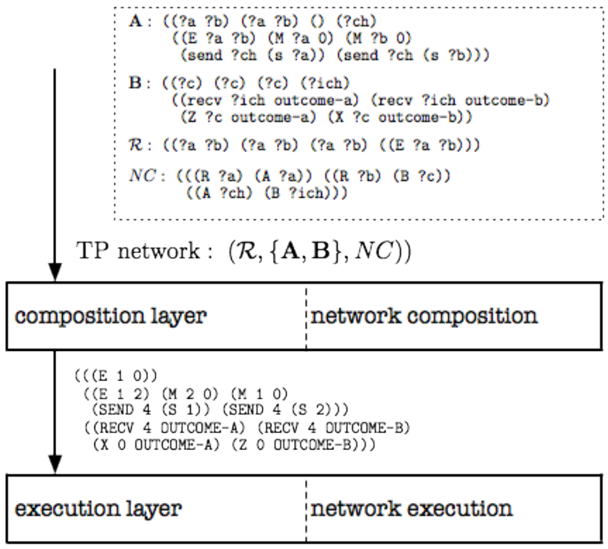
$\wedge\mathcal{X}$
composition layer
$\wedge\mathcal{X} : (\mathcal{I} \otimes \mathcal{H}) \wedge\mathcal{Z} (\mathcal{I} \otimes \mathcal{H})$
pattern composition
pattern assembly
execution layer
interpreter
$q_2, q_3 : |\psi_2\rangle \otimes |\psi_3\rangle$
quantum simulator

Figure 1: Schematic representation of the virtual MC framework, using a Controlled NOT pattern example.

the MC as presented in its original publication Danos *et al.* (2007). We identify the command sequence and measurement patterns as its two main features. The command sequence describes the actual quantum computation semantics, while the measurement pattern acts as an abstraction mechanism to compose multiple command sequences into a larger one.

5.2.1 Execution layer

The execution layer takes a concrete command sequence as an input and is in charge of performing the semantics. A command sequence is an expression that lists quantum operations with concrete qubit identifiers. We present these as integers, used as indices. This is expressed as structured data, for the computer to work with it. We choose to represent this data syntactically with *symbolic expressions* or *s-expressions* (McCarthy, 1960), as popularised by the programming language Lisp. At the same time we choose to adhere insofar as possible to the notation used in the formal model. A BNF specification of this syntax can be found in the implementation section 5.4. The following command sequence will after execution have performed the hadamard operation with qubit `1` as



```
A : ((?a ?b) (?a ?b) () (?ch)
     ((E ?a ?b) (M ?a 0) (M ?b 0)
      (send ?ch (s ?a)) (send ?ch (s ?b)))

B : ((?c) (?c) (?c) (?ich)
     ((recv ?ich outcome-a) (recv ?ich outcome-b)
      (Z ?c outcome-a) (X ?c outcome-b))

R : ((?a ?b) (?a ?b) (?a ?b) ((E ?a ?b)))

NC : (((R ?a) (A ?a)) ((R ?b) (B ?c))
       ((A ?ch) (B ?ich)))
```

TP network : $(\mathcal{R}, \{\mathbf{A}, \mathbf{B}\}, NC))$

```
(((E 1 0))
 ((E 1 2) (M 2 0) (M 1 0)
  (SEND 4 (S 1)) (SEND 4 (S 2)))
 ((RECV 4 OUTCOME-A) (RECV 4 OUTCOME-B)
  (X 0 OUTCOME-A) (Z 0 OUTCOME-B)))
```

Figure 2: Schematic representation of the virtual DMC framework, using a teleportation network example.

input and qubit 2 as output.

$$\texttt{((E 1 2) (M 1 0) (X 2 (s 1)))}\ . \tag{30}$$

An important difference here is that the command sequence *operations are executed from left to right.* In the formal model's notation this was the reverse. The right to left order reflects matrix operations in linear algebra. In a virtual model this makes less sense and so we choose a left to right ordering, out of both implementation reasons and computer science tradition.

The execution layer consists of an interpreter which directly executes each operation in turn. It is essentially an automatic version of the operational semantics described earlier in Sec. 3 and specified in full in the appendix. In the formal model the quantum state is represented as a single state vector, a tensor product of all qubits during initialisation. Given the computational explosion of operations on larger tensors, this is highly impractical in a classical simulation environment. In our implementation we use a technique that only puts qubits together in a tensor when entangled. This is possible because the MC ensures us that qubits only get entangled after the entanglement operation, and only measurement will again break this entanglement. This technique is described in further detail in the implementation section 5.4.

5.2.2 Composition layer

The language understood by our execution layer is in itself a full QC language, although writing larger quantum algorithms in it by hand quickly becomes a tedious and error-prone process. The measurement calculus introduces measurement patterns and pattern composition simplify this task. Pattern definitions enable local reasoning while composition allow for the creation of more complex programs out of smaller parts. More general patterns can be defined by using parameters, and algorithms can be programmed by composing them out of smaller patterns. All these abstractions are taken care of by the composition layer explained here. Its compiler effectively machine-generates command sequences for the execution layer.

Patterns are command sequences where all qubit names are categorised as either input, output or working qubits. Defining patterns is a straightforward process: the command sequence syntax is the same as the execution layer language, but instead of concrete qubit references, variable names may be used. We precede variable names with a question mark to stress the difference with constants and concrete identifiers. The measurement pattern for the familiar Hadamard pattern is expressed in our environment as

$$\mathcal{H} := \texttt{((?i ?o) (?i) (?o) ((E ?o ?i) (M ?o 0) (X ?o (s ?i))))}\ .$$

Additionally, a pattern definition can be parametrised, which is necessary for patterns such as the $\mathcal{J}(\alpha)$-pattern (Danos *et al.*, 2007). This is implemented in a straight forward way by using substitution, and does not influence other semantics.

Measurement patterns can be composed into a larger one in parallel or sequentially, as defined formally in Eqs. (2) and (3) respectively. These definitions rely on the author of the pattern to prepare the composition by rewriting qubit names and, in case of sequential composition, tensoring identity patterns where needed to make sure the condition $V_1 \cap V_2 = O_1 = I_2$ is met.

In a parallel composition, the patterns do not share any qubits. It is up to the author to choose unique qubit names in a way such they overlap, usually by renaming those that names do. In the following example we perform a parallel composition of two Hadamard patterns.

$$\begin{aligned}
\mathcal{H} &:= \texttt{((?i ?o) (?i) (?o)} \\
&\qquad \texttt{((E ?o ?i) (M ?o 0) (X ?o (s ?i))))} \\
\mathcal{H}(1,2)\circ\mathcal{H}(3,4) &:= \texttt{((1 2 3 4) (1 2) (3 4)} \\
&\qquad \texttt{((E 1 2) (M 1 0) (X 2 (s 1))} \\
&\qquad\ \ \texttt{(E 3 4) (M 3 0) (X 4 (s 3))))}
\end{aligned}$$

In a sequential composition the patterns have interdependencies; patterns intentionally share qubit names. The pattern author has to rename and match the correct qubit names to get the correct combinations. Using the same example, we sequentially compose two $\mathcal{H}$ patterns by matching one's output with the other's input.

$$\begin{aligned}
\mathcal{H}(1,2)\circ\mathcal{H}(2,3) &:= \texttt{((1 2 3) (1) (3)} \\
&\qquad \texttt{((E 1 2) (M 1 0) (X 2 (s 1))} \\
&\qquad\ \ \texttt{(E 2 3) (M 2 0) (X 3 (s 2))))}
\end{aligned}$$

It is crucial to automate this renaming process in a setting where we wish the programmer to be able to compose arbitrary patterns in intricate configurations. Our pattern compiler does precisely that. It allows the programmer to declare his intentions in non-trivial composition cases, whereas a standard rule is applied in absence thereof. This allows us to automate the tedious part of the qubit renaming process, while at the same time reducing the chance for human error on larger pattern compositions. In our approach the programmer has to list patterns in order and, in the non-trivial cases, give pairs of qubits to specify how patterns are to be composed. The patterns are composed by first instantiating the involved measurement patterns with fresh variable names, and then matching the names of the given variable pairs. Our *automated renaming process* works by generating a set of bindings that map the qubit variable names in every pattern to new ones, such that the chosen names are equal if they appear in the same pair. This is essentially the same process as one is assumed to execute manually when composing patterns in the formal model. For a more detailed and formal explanation of this renaming process we refer to the implementation section 5.4.

5.3 The distributed quantum virtual machine

Now that our virtual machine for sequential MC computations is established, we can describe its extension into a distributed version for the DMC. Incorporating distribution into our framework comes down to extending each of the abstraction layers. Our design philosophy has been to view network definitions as essentially a specialised forms of pattern composition. That is, they are dealt with in an extension of the composition layer from Sec. 5.2.2, and compiled towards a set of distributed pattern definitions, i.e. patterns which also allow communication commands. This set of distributed patterns, together with information on how the network is set up in terms of shared resources and agent channels agents, is then assembled to an executable form and sent to a distributed extension of the execution layer from Sec. 5.2. We discuss the execution layer in Sec. 5.3.1 and composition, now called network layer, in Sec. 5.3.2 below.

5.3.1 Execution layer

As with the MC case, the input language for the execution layer is executed by an interpreter. In the case of the DMC, a network with agent patterns is compiled into a list of command sequences, instead of a single one. Each of these sequences is the data representation of an agent's command sequence in the context of a distributed network. The DMC execution layer is to run these sequences concurrently. The shared resource pattern (behind the double bars $||$ in the formal model) is dealt with during a preparation step before starting the concurrent execution of the agents' command sequences. This way the shared qubits are initialised in the right way once we start the actual execution. While the agent abstraction is not explicitly present in the execution layer language, it does feature the required distribution functionality. Namely channel communication operations and concurrent execution of multiple command sequences.

The syntax used to express command sequences has changed little compared to the MC's case, adding a `send` and `recv` operation for message communication over a channel. The main difference is the DQVM's interpreter interleaving the execution of multiple command sequences, and handling (simulating) the channel communications. Full syntax, semantics and implementation details are provided in the implementation section 5.4.

5.3.2 Network layer

The network layer is the extended composition layer, where the programmer defines distributed programs. Such as, for example, the network for entanglement sharing specified formally in Eq. (9). A network consists of a list of agent definitions, a shared resource pattern, and a network configuration which specifies how resources are distributed and connected. Concretely, a network specifies a list of patterns, namely a single regular pattern for the shared resources and the extended patterns of all involved agents. For example, to define the entanglement swapping network from Eq. (9) we need the leader agent pattern **L** together with the agent pattern **A** and the resource pattern $\mathcal{R}$:

$$\begin{aligned}
\mathbf{L} &:= \big((?\mathtt{q}_0, ?\mathtt{q}_1, ?\mathtt{q}_2)\ (?\mathtt{ch}_0, ?\mathtt{ch}_1, ?\mathtt{ch}_2) \\
&\qquad (M^{GHZ}_{?q_0,?q_1,?q_2}\ (\mathtt{send}\ ?\mathtt{ch}_0\ (\mathtt{s}\ ?\mathtt{q}_0))\ (\mathtt{send}\ ?\mathtt{ch}_1\ (\mathtt{s}\ ?\mathtt{q}_1))\ (\mathtt{send}\ ?\mathtt{ch}_2\ (\mathtt{s}\ ?\mathtt{q}_2)))\big) \\
\mathbf{A} &:= \big((?\mathtt{q})\ (?\mathtt{ch})\ ((\mathtt{recv}\ ?\mathtt{ch}\ \mathtt{v})\ (\mathtt{X}\ ?\mathtt{q}\ \mathtt{v}))\big) \\
\mathcal{R} &:= \big(\mathtt{V}\quad \mathtt{I}\quad \mathtt{O}\quad ((\mathtt{E}\ ?\mathtt{a}\ ?\mathtt{b})\ (\mathtt{E}\ ?\mathtt{c}\ ?\mathtt{d})\ (\mathtt{E}\ ?\mathtt{e}\ ?\mathtt{f}))\big)\ ,
\end{aligned}$$

where `V = I = O = (?a ?b ?c ?d ?e ?f)`. The full network is specified by the expression $ES := (\mathcal{R}, \{\mathbf{L}, \mathbf{A}_{(1)}, \mathbf{A}_{(2)}, \mathbf{A}_{(3)}\}, NC)$, where NC is the network configuration.

$$\begin{aligned}
NC \quad = \quad \Big(&\big((\mathcal{R}\ ?\mathtt{a})\ (\mathbf{L}\ ?\mathtt{q}_0)\big)\ \big((\mathcal{R}\ ?\mathtt{c})\ (\mathbf{L}\ ?\mathtt{q}_1)\big)\ \big((\mathcal{R}\ ?\mathtt{e})\ (\mathbf{L}\ ?\mathtt{q}_2)\big) \\
&\big((\mathcal{R}\ ?\mathtt{b})\ (\mathbf{A}_{(1)}\ ?\mathtt{q})\big)\ \big((\mathcal{R}\ ?\mathtt{d})\ (\mathbf{A}_{(2)}\ ?\mathtt{q})\big)\ \big((\mathcal{R}\ ?\mathtt{f})\ (\mathbf{A}_{(3)}\ ?\mathtt{q})\big) \\
&\big((\mathbf{L}\ ?\mathtt{ch}_0)\ (\mathbf{A}_{(1)}\ ?\mathtt{q})\big)\ \big((\mathbf{L}\ ?\mathtt{ch}_1)\ (\mathbf{A}_{(2)}\ ?\mathtt{q})\big)\ \big((\mathbf{L}\ ?\mathtt{ch}_2)\ (\mathbf{A}_{(3)}\ ?\mathtt{q})\big)\ \Big)
\end{aligned}$$

As in Sec. 5.2.2, each of these patterns is instantiated with distinct qubit and channel variable names. The network configuration drives the composition of patterns in a network, which steers an automatic renaming process like in Sec. 5.2.2. This network configuration contains two elements. First, a list of qubit variable pairs linking output qubit variables of the shared resource pattern to input qubit variables of some agent pattern. Second, there are now also channel variables to be matched between agent patterns, due to the communication extension to their command sequence. However, the matching of channel names can be performed by the same renaming process defined earlier. After the automated renaming process the resource and agent patterns are assembled and collected in a list, forming the multiple command sequences that are used as input to the DQVM's interpreter in the execution layer.

5.4 Virtual DMC Syntax, Semantics and implementation

The following sections delve deeper into some of the implementation and design details of the virtual DMC framework. The structure follows that of the design:

execution and composition first for the MC case, then the same for the DMC extensions. Note that everything presented here has been implemented and is executable. Rather than listing Common Lisp source code, we expose the implementation in a semi-formal way.

5.4.1 MC Execution layer: Command Sequences

Syntax We first presents the syntax we have chosen to represent the concrete command sequences for the MC. We use a BNF notation where the symbols in boldface indicate syntax (round brackets and literals), while square and curly brackets are meta-syntax symbols which denote option (zero or one occurrence) and repetition (occur any finite number of times) respectively.

```
<sequence>       ::=  ( { <instruction> } )
<instruction>    ::=  <correction> | <measurement> | <entanglement>
<correction>     ::=  ( X <quref> [ <signal> ] ) |
                      ( Y <quref> [ <signal> ] )
<measurement>    ::=  ( M <quref> <angle> [ <signal> ] [ <signal> ] )
<entanglement>   ::=  ( E <quref> <quref> )
<signal>         ::=  0 | 1 | <input-name> |
                      ( s <quref> ) | ( + <signal> { signal } )
```
(31)

The only real difference with the formal command sequence syntax is the handling of signals of measurement outcomes, referring to them explicitly with `(s <qref>)` rather than relying on the naming convention s_n to denote the outcome on qubit n. This is the full syntax of the command sequence execution language we use as a 'quantum assembly language', it contains only concrete identifiers and constants.

Command Sequence Interpreter The interpreter is the actual implementation of the MC semantics, it takes a language in the above syntax and directly executes it one operation at a time, following the semantics in the formal model. We have chosen for a simple recursive implementation of the interpreter, where it continuously calls itself with the remaining command sequence after executing the first operation. Each recursive call of the interpreter takes, modifies and passes along the execution state. Like in the formal semantics this state consists of a quantum and classical part, respectively denoted Q and Γ below. They are defined as follows in our interpreter.

$$\Gamma ::= (o, i) \tag{32}$$

$$Q ::= T \,|\, T \otimes Q \tag{33}$$

$$T ::= (\{q_1, q_2, \ldots q_n\}, |\psi_{q_1 q_2 \ldots q_n}\rangle), n \in \mathbb{N} \tag{34}$$

Differences with the formal model arise from practical considerations. To save on space, the quantum state Q is represented as a set of *tangles* T. As described below, tangles achieve this by avoiding unnecessary tensor products. The classical state Γ is split into the outcome map o and the input map i to simplify lookup of qubit and input names respectively. Effectively, the outcome map contains only measurement results during execution. The input map contains other classical information, typically used to communicate and control the computation classically. This simplifies implementation and prevents unintended name clashes. As an illustration, consider the quantum cryptographic protocol by Bennett and Brassard... (1984) that requires a classical input to choose which basis to measure a fresh qubit in. The measurement result would be put in the outcome map, while the classical 0 or 1 signal to control the measurement basis is to be found in the input map.

Quantum state using Tangles The introduction of our *tangles* concept comes from the following observations. First, there is an exponential increase in space complexity when combining multiple qubits together using the tensor product. Next we see that in practice the entanglement graph is often not connected, meaning that a qubit is not always entangled with all other qubits in the graph. In the formal model, the quantum state is represented as a tensor product of all qubits in the entanglement graph out of convenience. Finally, the MC guarantees that only the Entanglement operation will entangle two qubits, forming an edge in the graph state. Indeed it is the only multiple-qubit operation. This is not obvious in the circuit model for quantum computation, where arbitrary multiple-qubit operations are common. Because of these reasons we introduce the concept of a tangle for our implementation of the quantum state. Conceptually, a tangle is a representation of a connected entanglement graph. It provides a more compact representation of the quantum state by avoiding unnecessary tensor products.

In our interpreter, the full qubit state Q is composed of a number of individual tangle objects T, which are, evidently, updated during execution. Concretely, during initialisation a new tangle $T_i : (\{q_i\}, |\psi\rangle)$ is created for each qubit, where q_i a single qubit reference, and auxiliary qubits have $|\psi\rangle = |+\rangle$. After interaction due to an entanglement operation $E_{q_i q_j}$ qubit references q_i and q_j are put into the same tangle $T_{ij\ldots} : (\{q_i, q_j, \ldots\}, |\psi_{q_i q_j \ldots}\rangle)$, while the original tangles T_i and T_j are destroyed. Here the dots represent other qubits which have interacted with qubits i or j earlier in the computation. After initialisation each command in the command sequence is executed closely following the semantics of the formal model, albeit updated to the slight variations in auxiliary structures as exhibited above. In our proof-of-concept implementation, we have implemented the semantics directly using numerical linear algebra. This implementation was based on a Lisp-based simulation environment developed earlier in (Desmet *et al.*, 2006).

5.4.2 Pattern composition and the automated renaming process

Qubit names in the formal MC have a double purpose, serving both as concrete qubit identifier and qubit variable by subjecting them to rewriting. In our virtual framework we enforce the use of qubit variables in measurement patterns, leaving the pattern compilation step to choose concrete qubit identifiers. The purpose of this is to enforce local reasoning: a qubit variable name is only valid inside the pattern currently being defined. When composing patterns sequentially, qubit names that have to be shared between patterns are provided separately using qubit variable pairs. The renaming process will match the correct variable names when composing the patterns.

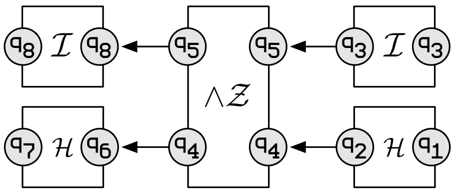


Figure 3: Graphical representation of $\wedge\mathcal{X}$'s pattern composition.

We explain the general composition case first, as we define the standard rule in terms thereof later on. As a running example, let us consider the Controlled-NOT pattern $\wedge\mathcal{X}$, which is defined in terms of the $\mathcal{H}$ and $\wedge\mathcal{Z} = (\{1,2\},\{1,2\},\{1,2\},E_{12})$ patterns, as follows.

$$\wedge\mathcal{X} := (\mathcal{I}(1)\otimes\mathcal{H}(3,4))\circ\wedge\mathcal{Z}(1,3)\circ(\mathcal{I}(1)\otimes\mathcal{H}(2,3)) \tag{35}$$

where $\mathcal{I}$ is the identity pattern, used as filler to match the number of in and output qubits. This composition pattern simply follows from the matrix identity. The same intent can also be expressed without using concrete qubit names as follows

$$\begin{aligned}\wedge\mathcal{X} := \{&(\mathcal{I}(q_8)\otimes\mathcal{H}(q_7,q_6))\circ\wedge\mathcal{Z}(q_5,q_4)\circ(\mathcal{I}(q_3)\otimes\mathcal{H}(q_2,q_1))\\ &\text{with } q_2=q_4, q_3=q_5, q_4=q_6, q_5=q_8)\},\end{aligned} \tag{36}$$

which is just the textual version of Figure 3 (in fact, as we shall see below, our framework also provides a graphical tool for specifying constraints, as in the figure). By simply matching variable names, we can derive Eq. (35) automatically from Eq. (36). In fact we can simplify Eq. (36) further by getting rid of identity patterns and only expressing non-trivial matching qubit names, which is specified in our (left-to-right) syntax as follows,

$$\{(\mathcal{H}(q_1,q_2)),\wedge\mathcal{Z}(q_4,q_5),\mathcal{H}(q_6,q_7)),\{(q_2,q_4),(q_4,q_6)\}\}\ . \tag{37}$$

Note that the syntax uses pairs for constraints, and we have reverted to our left-to-right evaluation order as everywhere in the virtual model. With this generalised notation for pattern composition in place we can derive an automatic renaming procedure, corresponding to the standard rules for composing patterns (e.g. linking up the first pattern's output qubits one by one with the second pattern's input qubits for sequential composition).

When a set of qubit pairs has been specified for a pattern composition – either automatically or by the user himself – the *automated renaming process* works by generating a set of bindings that map the qubit variable names in every pattern in the composition to new ones, such that the chosen names are equal if they appear in the same pair. This is essentially the same process as one is assumed to execute manually when composing patterns in the formal model. To ensure that the construction of new bindings occurs in a well-defined and finite manner, elementary compositions in a composite structure are processed in topological order. To be precise, an arbitrary pattern composition expression such as the one in Eq. (37), is viewed as a graph, with patterns as nodes and qubit variable pairs as edges. Since we allow only pattern combinations that form directed acyclic graphs, there is a unique topological ordering on the list of nodes, and this is the order in which we evaluate composition bindings. The full set of bindings B is constructed iteratively through the following rules on the pairs of every elementary pattern composition in topological order.

$$\frac{B(q) = B(q') = \varnothing \ , q_c \text{ fresh}}{(q, q') \to B[q_c/q][q_c/q']} \tag{38}$$

$$\frac{B(q) = q_c}{(q, q') \to B[q_c/q']} \tag{39}$$

$$\frac{B(q') = q_c}{(q, q') \to B[q_c/q]} \tag{40}$$

In other words, when both names in the pair do not appear in the binding list, rule (38) will trigger. A fresh qubit variable name q_c is chosen and added as binding for both variable names in the pair. Rules (39) & (40) ensure that if a binding already exists for one of the variable names in the pair, the other will use the same binding. The topological sort will ensure that at all times only one of the three rules will execute.

Once the binding set B is constructed qubit variables in each of the composing patterns are substituted with their bound value, $P'_i = \{B(q) | \forall q \in P_i\}$. After this renaming process all patterns in the composition can finally be merged pairwise by merging qubit sets and appending command sequences. Since our composition mechanism encompasses both definitions in the formal model our rules for joining qubits sets are also slightly more complicated (as they are more general). Concretely, we have the following definition, where patterns are assumed to have already passed the renaming process.

Definition 5.1 *The* composition *of patterns* $\mathcal{P}_1 = (V_1, I_1, O_1, \mathcal{A}_1)$ *and* $\mathcal{P}_2 = (V_2, I_2, O_2, \mathcal{A}_2)$ *is defined as the pattern* $\mathcal{P} = (V_1 \cup V_2, I, O, \mathcal{A}_1\mathcal{A}_2)$ *where*

$$I = I_1 \cup (I_2 \backslash O_1) \tag{41}$$
$$O = (O_1 \backslash I_2) \cup O_2 \ . \tag{42}$$

Indeed, qubit variables from the old input and output sets that were matched become auxiliary qubits and hence are only represented in the working set V.

We stress again that except in the most complex cases, the programmer does not have to explicitly denote qubit bindings when creating pattern compositions. Indeed, in most situations these can be created automatically or one can use implicit notation for qubit bindings as in the formal model. For example, we automatically derive regular sequential composition as specified in Eq. (2) when both patterns have distinct and unique variable names while the number of output and input qubits are the same. The automatic renaming process ensures that the condition $I_2 = O_1$ holds by automatically creating a composition expression of P_1 and P_2 while identifying each qubit in O_1 with the corresponding one in I_2 (i.e. in sequential order). Likewise, parallel composition as specified in Eq. (3) can be trivially derived, simply by not specifying any qubit pairs.

Without being renamed, the qubit names of each pattern in the parallel composition remain unique. Because of this, Eqs. (41)-(42) essentially merge the corresponding qubit spaces in the right way. With these shortcuts for the original composition operations in hand, we may now return to our example pattern $\wedge\mathcal{X}$. Indeed, while one way of specifying this composition is given in Eq. (37), by relying on the shortcut for sequential composition we can also express this composition without qubit names as:

$$\wedge\mathcal{X} := (\mathcal{I} \otimes \mathcal{H}) \circ \wedge\mathcal{Z} \circ (\mathcal{I} \otimes \mathcal{H}) \ . \tag{43}$$

The compiler will instantiate each pattern with fresh variable names, which, since they are unique, will precisely lead to the desired compositions as specified in the expressions above.

5.4.3 DQVM, the DMC execution layer

We extend the input language syntax for the QVM (specified in Eq. (31)) to support DMC computations as follows:

```
<network program>     ::=  ( <resources sequence>  { <agent sequence> } )
<resources sequence>  ::=  ( { <instruction> } )
<agent sequence>      ::=  ( { <agent instruction> } )
<agent instruction>   ::=  <instruction> |
                           <channel-send> | <channel-receive>
<channel-send>        ::=  ( send <channel-name> <signal> )
<channel-receive>     ::=  ( recv <channel-name> <input-name> ) .
                                                                    (44)
```

This is again a mirror of the formal syntax, bar the use of the labels `send` and `recv` instead of `!` and `?` to indicate message sending and reception respectively – this to avoid confusion with variable names. Also, communication commands carry a variable specifying the channel to be used for communication, an improvement over the formal model where the use of channels was somewhat implicit. Finally, preceding the agent sequences we see the command sequence to construct the shared resource. It is used as a kind of initialisation procedure, carried out before the rest of the program. This resource sequence is carried out by the regular MC execution layer as seen in Sec. 5.2. The quantum state resulting from this execution is then passed on to the initial state of the network extended execution layer.

The main functionality of the DQVM's interpreter over the QVM's is to do with communication operations and the scheduling of multiple command sequences. Practically, this means that the current state of execution retained in the interpreter's environment not only contains information on the quantum and classical states of the network (Q and Γ respectively) but also on issues to do with channel usage and scheduling. We call the latter the *network state* N and add it to the interpreter's environment. The network state $N = \{C, P\}$ holds the *channel map* C and the *network program* P. The latter is a simple collection of each agent's command sequence at each point in the computation, and is used by he interpreter to keep track of each command sequence as it is dynamically scheduled for execution. Each command sequence is executed to the point where it is empty or a send or receive operation blocks. A next command sequence will then take turn via a round robin selection and be executed in the same fashion. This round robin system ensures that sequences with blocking operations become available for execution at later time, successfully executing the send or receive operation in question if a matching receive or send has been performed by a previous sequence's turn. This process halts when all command sequences are empty. Deadlocks or similar errors will cause the interpreter to get stuck in a loop. We note that the formal semantics of a network is the same for any chosen schedule and for this reason we may choose a schedule at will without having to worry about the network's behaviour deviating from the intended one (Danos *et al.*, 2005).

The second component of the network state is the channel map C. It simply maps channel names to values (sent along the channel) and is used to implement the communication primitives in a straightforward way:

$$\frac{C(ch) = \varnothing \qquad \Gamma(s) = v}{\Gamma, C, (\texttt{(send } ch\ s\texttt{)}\ \mathcal{E}) \quad \rightarrow \quad \Gamma, C[v/ch], (\mathcal{E})} \tag{45}$$

$$\frac{C(ch) = v}{\Gamma, C, (\texttt{(recv } ch\ name\texttt{)}\ \mathcal{E}) \quad \rightarrow \quad \Gamma[v/name], C[\varnothing/ch], (\mathcal{E})}\ . \tag{46}$$

Here ch is a channel name, $\Gamma(s)$ denotes the value of the signal s with respect to the given outcome map and $\mathcal{E}$ is the rest of the execution sequence. In other words, the send operation associates the value to be sent to the appropriate channel name in the channel map C, while the corresponding receive operation removes the value from C and stores it in the classical state. In case where a channel already contains a value during a send operation along a particular channel, the send operation blocks, and likewise for a receive operation on an empty channel:

$$\frac{C(ch) = v}{C, (\texttt{(send } ch\ v'\texttt{)}\ \mathcal{E}) \quad \rightarrow \quad C, (\texttt{(send } ch\ v'\texttt{)}\ \mathcal{E})} \tag{47}$$

$$\frac{C(ch) = \varnothing}{C, (\texttt{(recv } ch\ name\texttt{)}\ \mathcal{E}) \quad \rightarrow \quad C, (\texttt{(recv } ch\ name\texttt{)}\ \mathcal{E})}\ . \tag{48}$$

Note that these rules constitute a concretisation of the semantical rule for classical communication in the formal model (see Eq. 10). Here we have chosen a concrete semantics using a form of *blocking*, in the sense that execution of a particular command sequence may halt until certain requirements are met, embodied in the equations above.

Next to the network state, the interpreter's environment also keeps track of the quantum and the classical state of the network. For practical reasons we reuse the original QVM's computation state as described in Sec. 5.2.1, i.e. we have one global quantum state Q and one global classical state Γ for the entire network. This simplifies the structure of the DQVM, allowing it to focus on the network-specific features while the QVM is used for the execution of regular MC operations. Indeed, the quantum state Q is passed through unchanged to the QVM which executes regular MC (i.e. quantum) operations. The execution schedule described above imposes a synchronised access to this global state (Q, Γ), while our naming procedure ensures that names are unique so that there are no clashes between classical variable names of different agents. Note that a more sophisticated implementation would adhere to the DMC's formal semantics where each agent has its own local state. For local quantum states this is to some extent captured by our use of tangles for disentangled quantum states (cfr. Eq. (32)). For classical states this would require a separate environment for each agent such that a received value is stored in the local classical state of the receiving agent.

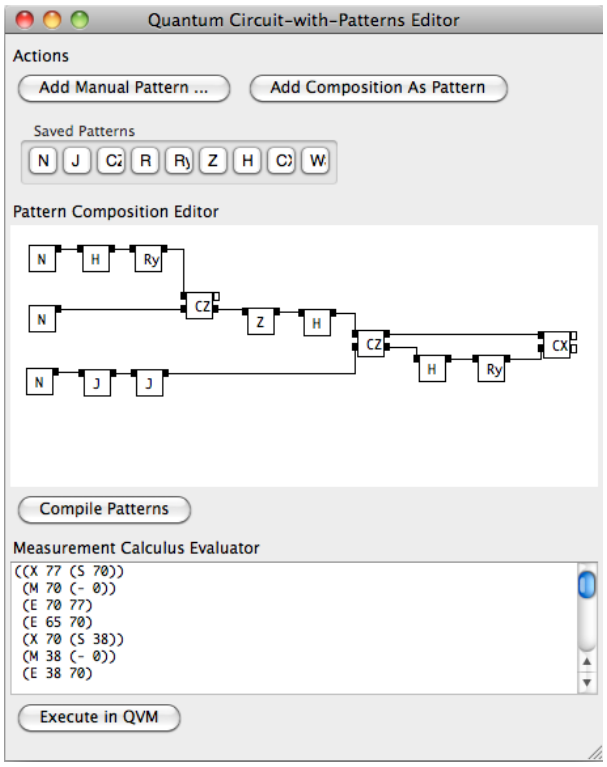
5.5 A graphical user interface

In the above we gave an overview on the design and architecture of our virtual machine for the DMC language. While the subject of this article has to do with the inner structure of our framework, the goal of the latter is nevertheless to provide a user-friendly programming environment. For this reason we have developed a graphical user interface (GUI) on top of the virtual machine to facilitate experimentation. This GUI tool currently supports only regular (non-distributed) patterns, allowing definition of patterns from scratch as well as in terms of compositions of existing patterns. Figure 4 shows the GUI being used to define a pattern, the W_3 entanglement pattern, as a composition of known patterns; at the bottom we find the compiled version of that same pattern. As seen previously in this section our framework allows for a general, more explicit form of composition, such that explicit set of qubit name pairs subsumes the implicit method of matching qubits by manual rewriting. While expressing these pairs in writing is still a feat for large programs, a graphical notation similar to Figure 3 is much more expressive. Our GUI uses a similar notation where black boxes denote in- and outputs and the user may connect these in arbitrary ways to concretise the desired connections. This technique allows a quantum programmer to express complex quantum algorithms more easily, certainly for composition structures with non-trivial connections between component patterns.

6 Conclusion

In the above we describe an assembly language for distributed measurement-based quantum (or DMC) computations in all its aspects. While the first half of this article deals with the formal model, the second half elaborates on a virtual framework developed in close relationship with the formal model, i.e. a programming environment for the DMC language.

DMC programs satisfy several formal properties crucial to the practical usability of the language, such as compositionality and context-freeness. We showed how to put these properties to use by formally implementing a composite program to control operations in a distributed setting, and demonstrated that the semantics does not change under the various composition operations. DMC was developed with expressiveness in mind, a crucial property when establishing the first layer in a distributed quantum programming paradigm. Indeed only through experimentation combined with abstraction can one hope to move towards a higher level in the language hierarchy. Our first experiments on paper already prove that DMC is indeed capable of expressing more complicated programs and that the formal features are necessary and sufficient in determining their functionality. This is very different in flavour from earlier formal


Quantum Circuit-with-Patterns Editor
Actions
Add Manual Pattern ...
Add Composition As Pattern
Saved Patterns
N
J
C:
R
R)
Z
H
C)
W:
Pattern Composition Editor
N
H
Ry
N
CZ
Z
H
N
J
J
CZ
H
Ry
CX
Compile Patterns
Measurement Calculus Evaluator
Execute in QVM

```
((X 77 (S 70))
 (M 70 (- 0))
 (E 70 77)
 (E 65 70)
 (X 70 (S 38))
 (M 38 (- 0))
 (E 38 70)
```

Figure 4: Graphical User Interface of the QVM's design tool, showing the composition and compilation of the 3-qubit W entanglement state preparation pattern.

frameworks in this area (Gay and Nagarajan, 2004; Jorrand and Lalire, 2005), which are much more concerned with issues such as verification, and focus on providing tools such as type systems to facilitate investigations of this nature.

While the first half or this article shows that it is possible in principle to write DMC programs within the formal model, the limits of its usability are acutely felt even for the relatively small example of distributed controlled operations. Indeed, as an experimentation platform our formal framework is only really useful in terms of a quantum virtual machine (QVM), a programming environment for the DMC language automating execution and composition of DMC programs. In the second half of this work we discuss a first implementation of such a virtual machine, a proof-of-concept execution environment effectively virtualising the formal DMC model. The benefits of having a QVM are many: as a first layer of a tiered quantum computation architecture (Svore *et al.*, 2006), a platform for automated verification and model-checking, and maybe most importantly, a basis from which to develop higher-order distributed quantum computation languages through experimentation and abstraction. Our QVM is built up in terms of several abstraction layers, most importantly a platform-independent execution layer to deal with the low-level semantics of basic patterns, and a composition layer to create and compose larger programs. The composition layer comes with an associated compiler that translates any compositional structure to one single pattern definition. Any pattern definition, be it specified directly by the user or produced by the compiler, is assembled into an expression that the execution layer understands, essentially a command sequence where all variable names have been replaced by concrete references in the intended way. While the execution layer is defined independently of any actual implementation, we used a Lisp-based simulation environment in our experiments to evaluate execution-layer expressions. The QVM is extended with distributed structures at all layers into a distributed quantum virtual machine (DQVM) which allows specification and execution of arbitrary distributed networks. Finally, a graphical user interface (GUI) is added to facilitate usability of the framework. We note that due to the fact that some aspects of the formal model necessitate further concretisation in a virtual model, there are some differences in syntax and semantics between the two. For example the implicit naming conventions for variables, channels, and in the composition of programs, required a concrete design in the virtual setting.

The virtual machine developed in this article, while covering almost all aspects of the formal model, is a first implementation and as such there are several avenues for improvement. We list these here moving from higher-abstraction layers to lower ones. First, the distributed layer requires several extensions to be fully compatible with the formal model, most importantly by giving agents a more prominent role and allowing network composition. In order to develop the GUI into a fully-featured development tool, it also needs to be lifted to a distributed setting: adding agents and networks to the graphical notation is top priority. Ultimately the goal is to make the graphical notation into a self-contained language, allowing the programmer to specify arbitrary programs without needing to create ex-nihilo patterns, which involves writing command sequence code by hand. Second, much work can be done at the level of optimisation of the execution layer, which now relies on conventional rather than optimal data structures for its implementation. We are in the process of developing tailor-built data structures, so enhancing the performance of our framework by relying on domain-specific measurement calculus optimisations. For example, we are currently looking into the stabiliser calculus as a means to efficiently execute MC operations where possible, switching to a different representation when operations unsupported by the stabiliser calculus are performed. Optimisations at the level of command sequences are also possible, such as detecting and directly initialising cluster states rather than constructing them incrementally by executing the required entanglements. On the other hand, classically simulating large entangled states means dealing with an exponential blow-up of computational time and space. For this reason the so-called EMC form (Danos *et al.*, 2007) (which puts entanglements first) is not always the preferred one in a simulation setting, since we need to minimise the size of entangled states during the length of the computation. Finding the right balance between direct cluster state generation and exploiting classical resources at their fullest is an exercise which is currently underway. Finally, we are heavily looking into parallel computing techniques to improve the simulation of quantum operations, at the moment carried out by straightforward linear algebraic techniques. Concretely, we are investigating the compilation of command sequences into a dataflow network (Gordon *et al.*, 2006). In such a network quantum states are represented by a long stream of amplitudes, which has the double benefit of exposing the inherent parallelism while at the same time relaxing the need to fit entire vectors inside the same computer memory. This line of research has already lead to a first implementation of a parallelised simulation environment in (Verhaegen, 2009).

7 Acknowledgements

This work is based on earlier research with V. Danos, E. Kashefi and P. Panangaden. The first author is supported by the Flemish Fund for Scientific Research (FWO) and the second by the CRYPTASC project, funded by the Institute for the encouragement of Scientific Research and Innovation of Brussels (IRSIB/I-WOIB).

8 Appendix

Here we give the full operational semantics of the distributed measurement calculus. We are concerned here with the small-step rules, indicating how atomic expressions in the language are evaluated. The big-step operational semantics of a program is found by grouping the pertaining small-step rules into one transition for the whole program. We use the standard notation of sequents and rules. A sequent $\Gamma \vdash E \downarrow$ is read as "given environment Γ, the expression E evaluates to the value v". In case that the environment itself changes during evaluation of an expression E we write $\Gamma, E \longrightarrow \Gamma'$. We write $\Gamma(x)$ for the value of x in Γ and $\Gamma[v/x]$ for the environment Γ with the added binding of x to v. Sequents can be combined into rules $\frac{S}{S'}$ which are just a different notation for $S \Rightarrow S'$.

There are four groups of rules, dealing with classical values (signals and angles), measurement patterns and distributed measurement patterns respectively. Each group builds on top of the previous one. The first group of rules is to do with the evaluation of signals.

$$\frac{}{\Gamma \vdash 0 \downarrow 0} \quad \text{and} \quad \frac{}{\Gamma \vdash 1 \downarrow 1} \tag{49}$$

$$\frac{}{\Gamma \vdash s_i \downarrow \Gamma(i)} \tag{50}$$

$$\frac{}{\Gamma[v/s_i] \vdash s_i \downarrow v} \quad \text{and} \quad \frac{}{\Gamma[v/s_i] \vdash s_j \downarrow \Gamma(j)} \quad \text{if } i \neq j \tag{51}$$

$$\frac{\Gamma \vdash s \downarrow v \quad \Gamma \vdash t \downarrow u}{\Gamma \vdash s + t \downarrow v \oplus u} \tag{52}$$

Here Γ is the outcome map or classical state, and $\oplus$ denotes addition in $\mathbb{Z}_2$. Angles, which can have signal dependencies percolated through via dependent measurements, are also purely classical. Values of signals are looked up in Γ via the ruleset for signals in order to determine the actual value of a measurement angle. This procedure is summarised in the following rules.

$$\frac{}{\Gamma \vdash \alpha \downarrow \alpha} \tag{53}$$

$$\frac{\Gamma \vdash s \downarrow v \quad \Gamma \vdash t \downarrow u}{\Gamma \vdash {}^t[\alpha]^s \downarrow (-1)^v.\alpha + u.\pi} \tag{54}$$

$$\frac{}{\Gamma \vdash [\alpha]^s \downarrow {}^0[\alpha]^s} \quad \text{and} \quad \frac{}{\Gamma \vdash {}^t[\alpha] \downarrow {}^t[\alpha]^0} \tag{55}$$

Having defined how signals and angles are evaluated, we can now move on to the operational semantics of the basic commands. These commands operate on an environment consisting of a quantum state ρ as well as a classical state Γ. We have presented these rules here for density matrices; in pure state derivations we often use state transitions for brevity. Specifically, for a pure state we have $\rho = |\psi\rangle\langle\psi|$, which is mapped to $L|\psi\rangle\langle\psi|L^\dagger$, with L any of the entanglement, Pauli or projection operators below. A pure state transition can then be alternatively specified as mapping $|\psi\rangle$ to $L|\psi\rangle$.

$$\frac{}{\rho, \Gamma, E_{ij} \longrightarrow \wedge Z_{ij} \rho \wedge Z_{ij}, \Gamma} \tag{56}$$

$$\frac{\Gamma \vdash s \downarrow v}{\rho, \Gamma, X_i^s \longrightarrow X_i^v \rho X_i^v, \Gamma} \tag{57}$$

$$\frac{\Gamma \vdash s \downarrow v}{\rho, \Gamma, Z_i^s \longrightarrow Z_i^v \rho Z_i^v, \Gamma} \tag{58}$$

$$\frac{\Gamma \vdash {}^t[\alpha]^s \downarrow \beta}{\rho, \Gamma, {}^t[M_i^\alpha]^s \longrightarrow_{\lambda_0} \langle +_\beta|_i \, \rho \, |+_\beta\rangle_i, \Gamma[0/i]} \quad , \lambda_0 = \frac{\mathrm{tr}(|+_\beta\rangle\langle +_\beta|_i \, \rho)}{\mathrm{tr}\rho} \tag{59}$$

$$\frac{\Gamma \vdash {}^t[\alpha]^s \downarrow \beta}{\rho, \Gamma, {}^t[M_i^\alpha]^s \longrightarrow_{\lambda_1} \langle -_\beta|_i \, \rho \, |-_\beta\rangle_i, \Gamma[1/i]} \quad , \lambda_1 = \frac{\mathrm{tr}(|-_\beta\rangle\langle -_\beta|_i \, \rho)}{\mathrm{tr}\rho} \tag{60}$$

$$\frac{\rho, \Gamma, C_1 \longrightarrow_\lambda \rho', \Gamma'}{\rho, \Gamma, C_2 C_1 \longrightarrow_\lambda \rho', \Gamma', C_2} \tag{61}$$

The first three commands are purely quantum and straightforward. The measurement command is the only command that affects the quantum state as well as the classical state. First, the measurement angle has to be evaluated, which in turn requires evaluating the X- and Z-signals by the previous sets of rules. Measurement commands are also the only nondeterministic commands, as the measured qubit is projected onto either $|+_\alpha\rangle$ or $|-_\alpha\rangle$ with transition probabilities as stated. Usually, the convention is to renormalise the state after measurement, but we do not adhere to it here, as in this way the probability of reaching a given state can be read off its norm, and moreover the overall treatment is simpler. The last rule is for a composition of commands.

Finally, we need a set of rules to deal with distributed extensions to patterns. Essentially these transitions describe how agents $\mathbf{A}(i, o) : Q.\mathcal{E}$ and networks $\mathcal{N} = |_i \mathbf{A}_i(\mathbf{i}_i, \mathbf{o}_i) : Q_i.\mathcal{E}_i \parallel \sigma$ evolve over different time steps. We adopt a shorthand notation for agents, leaving out classical inputs and output, which do not change with small-step reductions.

$$\begin{aligned}\mathbf{a}_i &= \mathbf{A}_i : Q_i.\mathcal{E}_i \\ \mathbf{a}_i.E &= \mathbf{A}_i : Q_i.[\mathcal{E}_i.E] \\ \mathbf{a}^{-q} &= \mathbf{A} : Q\backslash\{q\}.\mathcal{E} \\ \mathbf{a}^{+q} &= \mathbf{A} : Q \uplus \{q\}.\mathcal{E}[q/x]\ ,\end{aligned} \tag{62}$$

where E is some event, and $\mathcal{E}_i$ and $\mathcal{E}_i'$ are event sequences. A *configuration* is given by the system state σ together with a set of agent programs, and their states, specifically

$$\sigma, |_i\Gamma_i, \mathbf{a}_i = \sigma, \Gamma_1, \mathbf{a}_1 \mid \Gamma_2, \mathbf{a}_2 \mid \ \ldots \ \mid \Gamma_m, \mathbf{a}_m\ . \tag{63}$$

The small-step rules for configuration transitions, denoted $\Longrightarrow$, are specified below; we provide some explanations afterwards. When the system state is not changed in an evaluation step, we stress this by preceding a rule by $\sigma \vdash$.

$$\frac{\sigma, \mathcal{P}(V, I, O, \mathcal{A}), \Gamma \longrightarrow_\lambda \sigma', \Gamma'}{\sigma, \Gamma, \mathbf{A} : I \uplus R.[\mathcal{E}.\mathcal{P}] \Longrightarrow_\lambda \sigma', \Gamma', \mathbf{A} : O \uplus R.\mathcal{E}} \tag{64}$$

$$\frac{\Gamma_2(y) = v}{\sigma \vdash (\Gamma_1, \mathbf{a}_1.\mathsf{c}?x \mid \Gamma_2, \mathbf{a}_2.\mathsf{c}!y \Longrightarrow \Gamma_1[x \mapsto v], \mathbf{a}_1 \mid \Gamma_2, \mathbf{a}_2)} \tag{65}$$

$$\frac{}{\sigma \vdash (\Gamma_1, \mathbf{a}_1.\mathsf{qc}?x \mid \Gamma_2, \mathbf{a}_2.\mathsf{qc}!q \Longrightarrow \Gamma_1, \mathbf{a}_1^{+q} \mid \Gamma_2, \mathbf{a}_2^{-q})} \tag{66}$$

$$\frac{L \Longrightarrow_\lambda M}{L \mid N \Longrightarrow_\lambda M \mid N} \tag{67}$$

Implicit in these rules is a sequential composition rule, which ensures that all events in an agent's event sequence are executed one after the other. The first rule is for local operations; we have written the full pattern instead of only its command sequence here to make pattern input and output explicit. Because a pattern's big-step semantics is given by a probabilistic transition system described by $\longrightarrow$, we pick up a probability λ here. Furthermore, an agent changes its sort depending on pattern's output O. The next rule is for classical rendezvous and is straightforward. For quantum rendezvous, we need to substitute q for x in the event sequence of the receiving agent, and furthermore adapt qubit sorts. The last rule is a metarule, which is required to express that any of the other rules may fire in the context of a larger system. L and R stand for any of the possible left-, respectively right-hand sides of any of the previous rules, while L' is an arbitrary configuration. Note that we might need to rearrange terms in the parallel composition of agents in order to be able to apply the context rule. This can always be done since the order of agents in a configuration is arbitrary. In derivations of network execution, we often do not explicitly write reductions as specified by (67), but rather specify in which order the other rules fire for the network at hand. It is precisely in this last rule that introduces nondeterminism at the network level, that is, several agent transitions may be possible within the context of a network at the same time.

References

Adão, P. and Mateus, P. (2005). A process algebra for reasoning about quantum security. In P. Selinger, editor, *Proceedings of the 3rd Workshop on Quantum Programming Languages (QPL04)*, pages 3–20.

Bennett, C. and Brassard..., G. (1984). Quantum cryptography: Public key distribution and coin tossing. *Proceedings of IEEE International ...*.

Bose, S., Vedral, V., and Knight, P. L. (1998). Multiparticle generalization of entanglement swapping. *Phys. Rev. A*, **57**(2), 822–829.

Brock, J. and Ackerman, W. (1981). Scenarios: A model of non-determinate computation. *Formalizations of Programming Concepts*, **107**.

Danos, V., D'Hondt, E., Kashefi, E., and Panangaden, P. (2005). Distributed measurement-based quantum computation. In P. Selinger, editor, *Proceedings of the 3rd International Workshop on Quantum Programming Languages (QPL 2005)*, volume 170 of *ENTCS*, pages 73–94. quant-ph/0506070.

Danos, V., Kashefi, E., and Panangaden, P. (2007). The measurement calculus. *Journal of the ACM*, **54**(2). quant-ph/0704.1263v1.

Desmet, B., D'Hondt, E., Costanza, P., and D'Hondt, T. (2006). Simulation of quantum computations in Lisp. submitted to Lisp workshop at ECOOP.

D'Hondt, E. (2005). *Distributed quantum computation – A measurement-based approach*. Ph.D. thesis, Vrije Universiteit Brussel.

Gay, S. J. and Nagarajan, R. (2004). Communicating quantum processes. In P. Selinger, editor, *Proceedings of the 2nd Workshop on Quantum Programming Languages (QPL04)*, Turku, Finland. Turku Centre for Computer Science, TUCS General Publication No 33.

Gay, S. J., Nagarajan, R., and Papanikolaou, N. (2005). Probabilistic model-checking of quantum protocols.

Gordon, M., Thies, W., and Amarasinghe, S. (2006). Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGOPS Operating Systems Review*, **40**(5), 162.

Jorrand, P. and Lalire, M. (2005). Toward a quantum process algebra. In *Proceedings of the first conference on computing frontiers*, pages 111–119. ACM Press.

Lalire, M. and Jorrand, P. (2004). A process algebraic approach to concurrent and distributed quantum computation: operational semantics. In P. Selinger, editor, *Proceedings of the 2nd Workshop on Quantum Programming Languages (QPL04)*, Turku, Finland. Turku Centre for Computer Science, TUCS General Publication No 33.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, ... *Communications of the ACM*.

Nielsen, M. A. and Chuang, I. (2000). *Quantum computation and quantum information*. Cambridge university press.

Raedt, K. D., Michielsen, K., Raedt, H. D., Trieu, B., Arnold, G., Richter, M., Lippert, T., Watanabe, H., and Ito, N. (2006). Massive parallel quantum computer simulator. quant-ph/0608239.

Raussendorf, R., Browne, D. E., and Briegel, H. J. (2003). Measurement-based quantum computation on cluster states. *Phys. Rev. A*, **68**(2), 022312.

Selinger, P. (2003). Towards a quantum programming language. *Mathematical Structures in Computer Science*. to appear.

Svore, K. M., Aho, A. V., Cross, A. W., Chuang, I. L., and Markov, I. L. (2006). A layered software architecture for quantum computing design tools. *IEEE Computer*, **39**(1), 74–83.

Verhaegen, S. (2009). *Stream programming for quantum computing*. Master's thesis, Vrije Universiteit Brussel.

Yimsiriwattana, A. and Lomonaco, S. J. (2005). Generalized GHZ states and distributed quantum computing. *AMS Contemporary Mathematics*.

Zukowski, M., Zeilinger, A., Horne, M. A., and Ekert, A. (1993). "Event-ready detectors" Bell experiment via entanglement swapping. *Phys. Rev. Lett.*, **71**(26), 4287–4290.