The Matrix Model of Computation

Sergio Pissanetzky, Research Scientist. Member, IEEE.

PREPRINT

Abstract

The Matrix Model of Computation (MMC) is a new Turing-complete virtual machine that serves as a formal container for the structural representation and analysis of systems, and finds applications in diverse areas such as Business, Software Engineering and Theoretical Physics. The MMC is a unifying notion in Systemics. It allows uniform, interoperable techniques to be applied to systems in general, using a uniform, machine – interpretable representation that is amenable to formal mathematical manipulation.

In this paper we show the MMC equivalence with Deutsch's Universal Quantum Computer, argue that every finitely realizable physical system can be perfectly represented by an MMC model, and propose the MMC as a Universal Model of Computation. We introduce the canonical form of the MMC and present two Scope Constriction algorithms for module clustering. The algorithms refactor the model to reconcile the flow of data with the flow of control, and effectively create objects by partitioning the MMC model into highly cohesive, weakly coupled modules. Finally, we propose an MMC-centric system environment where the MMC serves as a formal tool for system analysis, and recommend an interdisciplinary effort to create it. In an accompanying paper, we present small examples in the areas of business, physics, UML models, and object-oriented analysis and design.

1 Introduction

The *Matrix Model of Computation* (MMC) was recently introduced as a Turing-equivalent virtual machine and a universal container for source code, and some of its applications to the analysis and evolution of software were described [1]. The MMC is a tuple of sparse matrices [2] $\mathcal{M} = (\mathsf{C}, \mathsf{Q})$, where C is the *matrix of services* and Q is the *matrix of sequences*. Matrix C describes a set of relations of various degrees. Each row of C corresponds to a tuple in one of the relations, called a *service*. Each column corresponds to a *domain* used in the relations and to a *variable* used in the services. A variable in the MMC has two properties, a *role* and a *value*. A variable can play the role of an argument (A), codomain (C), or mutator (M) in a service. Hence, the value of element C_{ij} is either A, C or M, depending on the role that variable j plays in service i. The element is left blank if variable j is not present in service i. A domain can be any set, such as the objects of a class, and a variable can represent one of the objects.

Matrix Q also describes a set of relations of various degrees. It defines the *sequences* in which the services are to be "executed". To each service in C there corresponds a relation in Q with the *conditional logic* that defines what service will follow, based on the values of 0 or more *control variables*. Each row of Q is a tuple in that relation, and establishes a link between the corresponding service and the service that follows, conditioned by the values of the control variables. The elements of Q are foreign keys identifying the previous (P) service and the following (F) service, and the values, not roles, of the control variables.

Since C and Q define many relations, it can be said that an MMC representation is a relational database. The representation is also a program, and it can run in an interpreter or be directly compiled into executable code [1]. But the MMC is not a programming language, although it can be made programmable if a data entry language is developed. The MMC is a formal mathematical entity, a transform, amenable to being operated upon by algorithms. Yet, it interfaces with source code via conversion tools. The MMC is Turing-complete, which guarantees that any program written in any language can be expressed in MMC format. Transformations applied to the MMC by algorithms induce corresponding transformations in the code it represents, hence the applications to software evolution. Figure 1 shows the MMC transforms for several elementary expressions.

[1] No compilers or interpreters have been developed yet.

expression	service	a	b	c
$a = b$	$=$	C	A	
$\omega\, a$	ω	M		
$a\, \omega'$	ω'	M		
$a = b\, \beta\, c$	β	C	A	A
$a = f(b, c)$	f	C	A	A

Figure 1: MMC transforms for some elementary expressions in C, where ω is any unary operator (such as $-$ or $++$), β is any binary operator, and a, b, c are variables. The conditional C expression $a = b\,?\,c : d$ would be described by the services $a = c$ and $a = d$, and two tuples in matrix Q with the values $true$ and $false$ for the control variable b.

In this paper we focus on the scope of the MMC. In Section 2 we consider the Universal Quantum Computer (UQC) introduced by David Deutsch [3] and on Deutsch's physical version of the Church-Turing principle:
Every finitely realizable physical system can be perfectly simulated by a universal model computer machine operating by finite means.
We demonstrate that any finitely specifiable UQC operating by finite means can be perfectly represented by a corresponding MMC, and therefore every finitely realizable physical system can be perfectly represented by an MMC. Furthermore, and since the UQC can simulate any other quantum computer with any desired degree of accuracy, we conclude that the MMC can as well represent any quantum computer with any desired degree of accuracy. This leads us to the conclusion that the MMC is a Universal Model of Computation.

Having established the universality of the MMC, we switch our attention to more practical, but still theoretical issues. Universality means that MMC algorithms are general system algorithms, not subject to any particular system, programming language or mathematical representation. In Section 3, we present the first two such algorithms, the Scope Constriction algorithms SCA1 and SCA2. The algorithms identify two features present in matrix C, the *scope* of the variables across the services, and the *data channel*, where data can be seen as flowing from the variables, along their scopes, and into the services that need it. The SCA algorithms *refactor* the matrix by applying symmetric channel-narrowing *permutations*. The result is a partition of C into strongly cohesive, weakly coupled clusters of services and domains, or *objects*. The objects can be separated into *classes* by submodelling, and the algorithms can be reapplied to what remains.

Actual applications of the MMC are considered in the accompanying paper [4]. They include an example where the SCA algorithms find a law of Physics, starting from a thought experiment, another where business rules are transformed directly to their MMC model, suggesting a direct path from business rules to executable code, and a third example where UML class models are created from the business MMC and analyzed in the context of the MMC. We believe that the MMC will find important applications in OO development and refactoring, UML, MDA, symbolic algebra, mathematical equations, theories of Physics, and other matters.

2 The Matrix Model of Computation and the Quantum Computer

In this Section we argue that the MMC can perfectly represent any finitely realizable physical system and propose the MMC as a Universal Model of Computation. The Universal Quantum Computer (UQC) was introduced by David Deutsch [3]. The UQC can perfectly simulate any finitely realizable physical system, including any Turing machine, any classical system, or any quantum system, while still operating by finite means. Since the UQC is fixed hardware, it is the *program* running on the UQC that controls the simulation. [2] We prove our argument by proving that the MMC can perfectly represent the UQC. Another more recent model, Adiabatic Quantum Computation, was found to be equivalent to UQC [5], therefore we need to refer only to the UQC. The UQC is an *abstract* machine, one that can, in principle, be physically realized but is not intended to be. The MMC, instead, is a *virtual* machine, a program that can represent an abstract machine but can run only on a real computer.

[2] 4-force systems are excluded. The 4 known forces in nature are gravitational, weak, strong, and electromagnetic, but there is no generally accepted theory for systems where all 4 forces intervene.

In a quantum computer, the *qubit* is the unit of information. A qubit is a *2-state observable*. It lives in a 2-dimensional *Hilbert* space, with two unit *eigenvectors* or *eigenstates* as the basis. The *spectrum* consists of the two corresponding *eigenvalues*, say 0 and 1. The *state* of the qubit is a linear *superposition* of the two eigenstates, in general with complex, irrational coefficients. The state itself can not be *observed*, but if the qubit is observed (read), it will yield one of the eigenvalues, 0 or 1, as the result. The spectrum corresponds to a regular bit, and the eigenvalues to its two possible values, 0 or 1, but the fact that the state is a superposition of eigenstates makes a quantum computer fundamentally different from a Turing machine.

The structure of the UQC, however, is similar to that of the Turing machine. The processor is a finite sequence of qubits. One of them, say the leftmost, is used by the UQC to actively signal when it has halted. The memory is an infinite sequence of qubits. An additional observable, say x, with a nominally unlimited spectrum $\{\ldots,-1,0,1,\ldots\}$, indicates the position in memory. These three elements correspond to the head, tape, and tape position of the Turing machine.

The UQC lives in a multi-dimensional Hilbert space, where each basis eigenstate is a normalized combination of the eigenstates of all the qubits and the observable x. The state of the UQC at time t, say $\big|\Psi(t)\big\rangle$, is a unit vector in that space, expressed as a linear superposition of the basis eigenstates. If the UQC is observed, it will yield one of its eigenvalues as the result, a combination of 0's and 1's.

The UQC computes in steps, just like a Turing machine. In the Schröedinger picture of Quantum Mechanics, each step involves a *transition* from one state $\big|\Psi(t)\big\rangle$ to another $\big|\Psi(t')\big\rangle$, and can be summarized by a constant, step-dependent unitary operator on the Hilbert space, say U:

$$\big|\Psi(t')\big\rangle = \mathsf{U}\,\big|\Psi(t)\big\rangle \tag{1}$$

where we have used the traditional bra-ket notation "$|\ \rangle$" for states. The UQC is a Turing machine if the dynamics guarantees that state $\big|\Psi(t')\big\rangle$ is an eigenstate provided that state $\big|\Psi(t)\big\rangle$ is one, i.e., if computation takes the UQC from eigenstate to eigenstate. The UQC can, however, evolve into states that are linear superpositions of eigenstates. UQC computation proceeds as follows:

Step 1. Prepare the UQC in an eigenstate $\Psi(0)$, where the memory is initialized with the image of some program P and input I, both consisting of a finite sequence of bits. The position x, the halt qubit h, the qubits in the processor p and the rest of the memory m are set to 0. A set of qubits is set aside for the "result" R. Together, P and I define the operator U and thus the dynamics of the UQC.
Step 2. The halt qubit is measured and examined.
Step 3. If the halt qubit is false, measure the result R and exit.
Step 4. If the halt qubit is true, evolve the UQC once, as in Equation (1).
Step 5. Go to Step 2.

Theorem 1. *Any finitely specifiable Universal Quantum Computer operating by finite means can be perfectly represented by a Matrix Model of Computation.*
Proof. The proof is by construction. We construct an MMC model that can represent any UQC. The model consists, as usual of the matrices C and Q. Using previously defined nomenclature [1]:

$\mathsf{C} =$

service	x	P	I	R	p	m	$\big\|\Psi\big\rangle$	U	h
`init`	C	A	A	C	C	C	C	C	C
`evolve`							M	A	
`loop`							A		C
`read`				C			A		

$\mathsf{Q} =$

actor	P	h	F
1			`init`
	`init`		`loop`
	`loop`	true	`read`
	`loop`	false	`evolve`
	`evolve`		`loop`

(2)

C contains 4 services. Service `init` receives the given program P and input I, initializes everything else to 0, prepares the UQC in the initial state $\big|\Psi(0)\big\rangle$, and defines operator U. Service `evolve` performs one step of computation, as indicated by Equation (1), and updates the state of the UQC. Service `loop` reads the halt qubit

h from the current state function. Service `read` reads result R from the state function. An *actor* in matrix Q invokes service `init`. This action represents Step 1 of UQC computation. Immediately after, service `loop` is called to read the halt qubit h, representing Step 2. Then, if h is true, `read` reads R and exits, representing Step 3. Else, when h is false, `evolve` is executed, representing Step 4, and control returns to `loop`, as in Step 5. This completes the proof.

Theorem 2. *Every finitely realizable physical system can be perfectly represented by a Matrix Model of Computation.*
Proof. Deutsch's physical version of the Church-Turing principle states that every finitely realizable physical system can be perfectly simulated by a finitely specifiable UQC operating by finite means. By Theorem 1, such a UQC can be perfectly represented by a Matrix Model of Computation. The theorem includes any Turing machine.

Theorem 3. Universality of the MMC. *Any quantum computer can be represented with arbitrary precision by a Matrix Model of Computation.*
Proof. A quantum computer is an idealized, theoretically permitted, finitely specifiable model, not necessarily a physically realizable system. Although the UQC itself has an infinite – dimensional state space, if can simulate with arbitrary precision any other quantum computer, still using a finite number of finite-dimensional unitary transformations at every step to simulate its evolution, and still remaining finitely specifiable. By Theorem 1, any finitely specifiable Universal Quantum Computer operating by finite means can be perfectly represented by a Matrix Model of Computation. It is in this sense that the MMC is a Universal Model of Computation.

3 The Scope Constriction algorithms

In this Section we introduce the *Scope Constriction Algorithms* SCA1 and SCA2. The algorithms operate on the *canonical* form of a *linear submatrix* of matrix C. They attempt to reconcile the flow of data and the flow of control by "constricting" the scope of the variables in bulk. The constriction effect is obtained by means of symmetric permutations that reduce the profile of the submatrix but do not affect the canonical form. The permutations *refactor* the submatrix, create highly cohesive, weakly coupled clusters of services and domains, and determine a partition of the submatrix into objects. Examples of application are presented elsewhere [4].

Refactoring frequently involves moving methods (services) from one class to another in order to increase cohesion in methods and objects and reduce coupling between objects. Refactoring attempts to improve code understandability by encapsulating behavior (services) and properties (domains) and reducing the number of variable scopes that cross boundaries between objects or methods ("constricting" the scopes). The SCA algorithms do all that, suggesting that these seemingly different processes are one and the same, and are indeed at the very root of the concept of object. These ideas are the foundation of a new concept in refactoring. The algorithms can be implemented as programs and installed on the MMC itself, making it self-refactoring, a program that constantly refactors itself and creates its own objects. The SCA algorithms can also be used in reverse, in combination with the reinsertion of submodels, to create efficient, highly linked code prior to compilation.

A directed *control flow graph* (CFG) $G = (V, E)$ can be associated with any MMC, where a vertex represents a service in C and an edge a tuple in Q. A path in the graph represents a possible flow of control. The set of services on the path is said to be *linear* if the path has no additional incoming or outgoing edges. The linear submatrix is obtained by removing from C all services not in the set and all domains not used by the services. Note that the services themselves do not have to be linear, they can contain submodels of any degree of complexity.

Many MMC models contain linear submatrices. For convenience in presentation we shall assume that the entire C is linear, and that its rows have been physically permuted in the order of the vertices of the path.

The canonical form of the Matrix of Services C

A service i is said to be a *constructor* for domain j when $\mathsf{C}_{i,j} = C$. The *span of a domain* is the set of services that share a non-blank element with the domain, and the *span of a service* is the set of domains that share a non-blank element with the service. In general, C is a rectangular matrix, because a service can construct many domains and a domain can have many constructors and mutators, but the following simple transformations can bring C first to a *linear* form, then to a square *canonical form*:

- If any variables used in C are initialized outside C, replace them with new local variables (similar to dummy arguments in a function call).
- If a service has $n > 1$ codomains, expand it into n similar services that return one codomain at a time.
- If a variable is mutated or assigned to more than once, introduce new local variables as needed.

The properties of the linear matrix C now are:

- Every domain has exactly one constructor and no mutators, and every service constructs exactly one domain (*single-assignment form*).
- A one-one correspondence between services and domains has been established.
- Matrix C is now square.
- Every row contains exactly one C and 0 or more A's on either side of the C, and every column contains exactly one C and 0 or more A's below the C.
- The services are physically ordered the same as the vertices of the linear path.

In the MMC, an *object* is an association of a set of services and a set of domains. The one-one correspondence between services and domains is such an association. Therefore, a domain and its constructor form an *elementary object*. The SCA algorithms encapsulate the elementary objects to form larger, more meaningful objects. The objects can be separated by submodelling, and the SCA algorithms reapplied.

The canonical form of C is finally obtained by permuting the columns in such a way that the C's are brought to the diagonal. Columns can always be arbitrarily permuted. The canonical form is a linear matrix, and has the following additional properties:

- The matrix is square lower triangular.
- The diagonal is full with all its elements equal to C.
- The lower triangle contains only A's and blanks, the upper triangle is empty.
- Rows i and $i+1$ commute if and only if $C_{i+1,i}$ is blank.

Service and sequence commutation

Commuting two adjacent services means reversing their order without affecting the overall behavior. Commutation is legal if and only if it does not reverse the order of initialization and use of any variable. In the canonical form, this means that an A in a column always commutes with another A but never with a C in the same column. A service in row i can be shifted up to any row $i' < i$ by means of repeated commutations. If the rightmost A in row i is in column j, the shift is legal provided the A stays below the C in column j, that is $i' \geq i_{\min} = j + 1$. If the uppermost A in column i is in row k, the service can be shifted down to a row $i'' > i$ if the C in column i stays above that A, or $i'' \leq i_{\max} = k - 1$. The interval $(i_{\min}, i_{\max})$ of width $k - j - 1$ is the *range of commutation* for service i.

There is a form of commutation between control flow branches in matrix Q that depend on two or more control variables, with important consequences when single- or multiple-inherited polymorphism (SIP, MIP) are used for implementation. Consider the following equivalent C programs:

Program A	**Program B**	**Program C**
`if(a && b)s1; if(a && !b)s2;`	`if(a){if(b)s1; else s2;}`	`if(b){if(a)s1; else s3;}`
`if(!a && b)s3; if(!a && !b)s4;`	`else {if(b)s3; else s4;}`	`else {if(a)s2; else s4;}`

In Program A, there exists a symmetry between variables a or b. The symmetry is preserved if Program A is

implemented using MIP. If there are n_a possible values of a, and n_b of b, a total of $n_a + n_b$ base classes are required. An upgrade to a new value of a, for example, would require just one new class. A SIP implementation that preserves symmetry is also possible if a and b are combined into a composite domain (with $n_a n_b$ values), but this would require $n_a n_b$ different base classes, and an upgrade to a new value of a would require n_b new classes. This is prohibitive if $n_a, n_b > 2$. The forms B, C, instead, discriminate one condition at a time, forcing an artificial order between the control variables and breaking the symmetry. The result is a very large number of choices, none of them very good. Consider a case with n control variables with 2 possible values each. There are $n!$ different possible orderings among the n variables. If either SIP or conditionals are used for implementation, there are 2^{2^n-1} possible combinations for each ordering, for a total of $n!\,2^{2^n-1}$. For example, there are 16 ways of coding a link with 2 control variables, and 768 for one with 3 control variables, and this is just one link. This is a good example of the support that the MMC can provide for software analysis. An example [4, Section 4] expands on the matter.

Since *refactoring* is a behavior-preserving transformation, commutation is an element of refactoring. Repeated commutations produce advanced refactorings. The concept of combining refactoring elements has been in the literature for quite some time. Authors have developed procedures to determine what to refactor, often based on software metrics, and tools that effect the refactorings. To our knowledge, however, the SCA algorithms discussed below are the first language-independent, formal algorithms that can systematically apply refactoring elements selected by the algorithm itself, improve several metrics at the same time, and create objects by encapsulating services and domains. This is a new concept in refactoring.

The scope of variables and the data channel

In the canonical C, column j represents the life cycle of variable j. It begins with C on the diagonal, where variable j is initialized, and ends with a *terminal* A, where variable j is used as an argument for the last time. The *scope* of the variable is the set of elements comprised between the initializing C and the terminal A, including the A but not the C. A variable is *active* for all services in the range of its scope. If the terminal A is in row k_j, the cardinality or *length* of the scope is $k_j - j$. The collection of scopes is called the vertical *envelope* of C [2, p. 15]. The envelope can be viewed as a *data flow channel*, comprised between the diagonal of C's and the terminating A's. Data flows from the initial C, where a variable is initialized, to the various A's in that column, and from there continues its flow to the services that use the A's and initialize other variables. The size of the envelope is related to the width of the data channel. A large envelope means a wide data channel, which may be undesirable. The accumulated length of scopes:

$$L(\mathsf{C}) = \sum_j (k_j - j) \tag{3}$$

happens to be the vertical *profile* of C [2, p. 15]. The maximum and average width of the data channel are defined as, respectively:

$$W_m(\mathsf{C}) = \max_j (k_j - j) \tag{4}$$

$$W_a(\mathsf{C}) = L(\mathsf{C})/n \tag{5}$$

where n is the order of C. We hereby propose these three quantities, L, W_m, and W_a, as metrics for linear sections of OO code. They apply equally well to mathematical equations, or to anything that can be modeled with the MMC. The metrics are illustrated and further discussed in the examples given in the accompanying paper [4].

The SCA algorithms discussed below reduce the profile by permuting services. Their effect is to reconcile the flow of data with the flow of control by narrowing the data channel, within constraints. They narrow the channel in the average, and usually also reduce $W_m(\mathsf{C})$. In a canonical C, all symmetric permutations

$$\mathsf{C}' = \mathsf{P}^T\ \mathsf{C}\ \mathsf{P}, \tag{6}$$

where P is a permutation matrix, are legal behavior-preserving refactorings provided they don't break the rules of commutation. The SCA algorithms determine the permutation P in such a way that $L(\mathsf{C})$ is reduced.

In general, it is better to define variables near the point where they are used. But service commutation constraints imposed by the logic in the program limit our ability to reorder the services, and we can not reduce the scope of a variable without affecting the scopes of others. Therefore, we have a coupled optimization problem that must be addressed as a whole: minimize $L(\mathsf{C})$ subject to the commutation constraints. We refer to this optimization as a *constriction* of the scopes, and we like to imagine it as "pressure" applied by an external agent to the profile as a whole, stressing the constraints and causing the services to yield to the pressure and regroup the best they can.

Under constriction, some individual scopes may grow, but the overall effect is to reduce the span of domains. This causes service-domain pairs to cluster as much as they can, increasing cohesion inside the clusters and reducing coupling between clusters. The net effect is to reconcile the flow of data with the flow of control in the program, and to create *objects* accordingly.

Algorithm SCA1

SCA1 is a graph algorithm, but it can also operate directly on the canonical matrix C. It attempts to reduce the profile of C by following a depth-first/shortest-path strategy in the CFG associated with C, and renumbering the rows accordingly, subject to the rules of service commutation. The idea is to push the A's as close to the diagonal as possible. The algorithm does not guarantee a true minimum, and the solution may not be unique, but simple experiments have yielded useful results [4]. The algorithm uses a set of spanning trees rooted at the sources (a source is a vertex with no incoming edges):

Step 1. Given matrix C of order n in canonical form, construct the directed CFG $G = (V, E)$. In G, a vertex corresponds to a service-domain pair, and an edge to an A in the lower triangle of C. The graph is acyclic.
Step 2. Mark all vertices as unnumbered. A vertex can be numbered only if it is unnumbered and not adjacent to an unnumbered vertex (v_j is adjacent to v_i if $vi \rightarrow v_j$ is an edge). This guarantees compliance with the rules of commutation.
Step 3. Select an unnumbered source and create a spanning tree rooted at it. The tree ends at sinks (a sink is a vertex with no outgoing edges) or at vertices that can not be numbered. If no more sources are available, go to step 6.
Step 4. Number the vertices starting from the root and following the shortest branch. Then backtrack to the previous vertex and repeat until the tree is exhausted.
Step 5. Go to step 3.
Step 6. Reverse the order and permute the matrix accordingly.

Once the matrix is permuted, the new objects should be identified and separated as submodels. Traditional pattern matching techniques are used to detect classes. Examples are given in the accompanying paper []. Once the classes have been identified, the matrix can be partitioned and the partitions separated as submodels, resulting in a further reduction of the profile. New linear sections that include the new objects can then be detected, and the algorithm reapplied to create more objects of a higher level.

Algorithm SCA2

SCA2 is a matrix algorithm. It attempts to reduce the profile of C by systematically shifting services within their range of commutation until no more legal shifts are possible. The algorithm does not guarantee a true minimum, or a unique solution. We conjecture that SCA2 resembles the process that happens in the mind of an analyst who is manually refactoring OO code, or a mathematician who is factorizing equations. The *first order* SCA2 algorithm is:

Step 1. Visit all services in some arbitrary order.

Step 2. For each service, determine its range of commutation, calculate the profile of C for each shift in the range, and effect the shift that results in the minimum profile.
Step 3. Repeat the previous steps until no more profile reductions are possible.

We say that C is *SCA2-locked* in the first order when Step 3 is satisfied. The *second order* SCA2 algorithm attempts to further reduce the profile by shifting *sets* of two or more adjacent services as a whole.

4 Conclusions and outlook

The MMC can perfectly describe any finitely realizable physical system, in full detail and without loosing any information. MMC services correspond to actions that the system can perform, and MMC sequences describe the logic of decision-making. This is in sharp contrast with many other substantially narrative system descriptions found in practice.

In this paper we have proposed the concept that services and sequences, as defined in MMC theory, are sufficient to describe or analyze any physical system. The representations can be interpreted by machine and transformed by mathematical operations or computer algorithms. Transformations performed on the MMC represent transformations on the system, and properties or features found in the MMC represent features of the system. The representations are independent, but they can interoperate provided they contain domains in common.

The question remains, of course, whether MMC representations are practical. We already know, in the case of computer software, that their size is comparable to that of the program [1]. Sparse matrix representations are very compact. But only experimentation can answer that question more in general. We begin addressing this matter with several small examples presented in the accompanying paper [4].

To conclude, we'd like to propose an MMC-centric system environment, as we did before for an MMC-centric software development environment [1]. Systems in the environment would have their own, independent but compatible MMC models. Since systems are very diverse, and the current state of development of the MMC is very limited, a large interdisciplinary effort will be needed. But the MMC is one, and the effort needs to be done only once, and then expanded as needed. With the addition of a suitable search engine, the MMC will become a multi-capability formal tool for the analysis of systems.

References

[1] Sergio Pissanetzky. "A relational virtual machine for program evolution". Proc. 2007 Int. Conf. on Software Engineering Research and Practice, Las Vegas, NV, USA, pp. 144-150, June 2007. The Matrix Model of Computation was introduced in this publication with the name Relational Model of Computation, but was later renamed because of a conflict.
[2] Sergio Pissanetzky. "Sparse Matrix Technology." Academic Press, London, 1984. Russian translation: MIR, Moscow, 1988.
[3] David Deutsch. "Quantum theory, the Church-Turing principle and the universal quantum computer." Proc. Royal Soc. London, vol A 400, pp. 97-117 (1985).
[4] Sergio Pissanetzky. Applications of the Matrix Model of Computation. Proc. 12th World Multi-Conference on Systemics, Cybernetics and Informatics: WM-SCI '08 Orlando, Florida, USA, June 29 - July 2, 2008.
[5] Ari Mizel, Daniel A. Lidar, and Morgan Mitchel. "Simple Proof of Equivalence between Adiabatic Quantum Computation and the Circuit Model." Phys. Review Letters, Vol. 99, 070502 (Aug 2007).

Acknowledgements. To Prof. Dr. Angelo Ferrari (SFCC, Gainesville, FL) and Dr. Peter Thieberger (BNL, NY) for contributing irreplaceable encouragement, and much more.