Applications of the Matrix Model of Computation

Sergio Pissanetzky, Research Scientist. Member, IEEE.
Sergio@SciControls.com

World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI'08)
PREPRINT

Abstract

The Matrix Model of Computation (MMC) is a new Turing-equivalent and Deutsch-equivalent virtual machine used for the mathematical analysis of systems. The importance of the MMC stems from the fact that it allows uniform techniques to be applied to all kinds of systems and makes them amenable to formal mathematical manipulation.

But the strength of the MMC lies in its applications. In this paper we use simple examples to illustrate three very different applications. First, in Physics, we examine a thought experiment in Newtonian mechanics, and discuss the effect of the Scope Constriction algorithm. The algorithm factorizes the equations and discovers the law of independence of the components of motion. We believe that similar ideas can be applied to search for fundamental symmetries in the theories of Physics, compare theories, or test new hypotheses.

Second, in business, we note that business rules are semantically similar to the MMC and sketch some ideas for their direct conversion to MMC format. We argue that much of the procedure can be automated, based on the fact that an MMC representation is a relational database, and many businesses already use relational databases. Since the representation is also a program and can be directly compiled, this offers the exciting possibility of a direct path from business rules to executable code without using any programming language.

Lastly, in Software Engineering, we examine relationships between the MMC and UML class models, MMC support for encapsulation, inheritance and polymorphism, and the use of the MMC for refactoring object-oriented code. Some analysis of the refactorings also illustrates the use of the MMC as an analytical tool for systems.

1 Introduction

The Matrix Model of Computation (MMC) is a recently introduced [1] Turing-equivalent and Deutsch-equivalent [2] virtual machine and container for complex systems, with applications to systems such as computer programs, theories of Physics, and businesses. A *container* is a computer implementation of a standard structure designed to support algorithms that operate on that structure and allow them to interoperate, based on a *model* of sufficient generality. *Structure* arises naturally when *regularities* are found in otherwise irregular data. Structure is what identifies elements present in the data, reveals their relationships, provides common ground for algorithms, reduces the overall complexity and makes data more manageable and understandable. Encapsulated regularities give rise to the concept of *object*. The objects are, in turn, data amenable to a further extraction of regularities and to further encapsulation into higher level objects. Bunge [3] has provided the ontological basis for these concepts, and Chidamber and Kemerer [4] and other authors have provided metrics.

In this paper we use the term *refactoring* to refer to the process of identifying regularities and encapsulating them into objects. The term was introduced in 1992 [5] in connection with object-oriented software, but authors subsequently extended it to the C language [6], which is not object-oriented, and even to DNA [7].

The importance of refactoring has been widely recognized in Software Engineering, and research has been intense for more than a decade. Yet, not many automated tools are available, and the ones that are lack interoperability and are language-specific. Most refactoring is still done manually by developers.

Refactoring is also present in the process of creating a theory of Physics. A *model* is first created to explain observations. The regularity is that data calculated by the model fall within the intervals predicted by the experimental errors, and the reduction in complexity is that the errors have been factored out and calculated results are independent of errors. The structure is now equivalent to that of a thought experiment, and is certainly more manageable and understandable than the original observations. More regularities may be present in the model, such as symmetries, and further refactoring steps may be necessary to find them before the theory is complete. Refactoring in Physics is done mostly manually by physicists, sometimes with the help of tools such as programs for symbolic algebra. But these, just as in Software Engineering, rely on ad-hoc representations of the system and have limited interoperability.

In this paper we present small examples of three very different applications. The first application is a thought experiment in Newtonian mechanics that describes one time step in the simulation of the motion of a mass particle subject to a force, in three dimensions. Application of the Scope Constriction Algorithm (SCA) [2] creates a class of objects that could be called "component of motion", and finds three objects of that class, thus recognizing the law of independence of the components of motion.

The second example applies to business software. There is only a narrow semantic gap between business rules or technical literature such as "how to" manuals, and the MMC. Direct transformation between the two is possible, particularly in the case of business rules based on the fact that the MMC is a relational database and many businesses already use relational databases for their data. It should be possible to automate the transformation, at least in part, because stored procedures and queries are programs and can be easily converted to the MMC. Since the MMC can be compiled into executable code, this suggests a direct path from business rules via the MMC to an executable program, without using any programming language.

Our aim is to emphasize the breadth of applications of the MMC, illustrate its mechanics, encourage other authors to develop algorithms for the MMC, and establish it as a mathematical tool for the analysis of systems in general.

2 An example in Newtonian Mechanics

This example describes one time step in the simulation of the motion of a mass particle under the action of a force in 3 dimensions, using a thought experiment based on Newton's 2nd. law. For convenience in presentation, the equations are written as code in a "C-like" language and in a linear "pre-canonical" form, with the required local variables already in place and a single-codomain service per line [2, Sec. 3]. The code has been intentionally scrambled to illustrate that SCA works independently of initial conditions:

Program P1

$$\begin{array}{rlcl@{\qquad}rlcl@{\qquad}rlcl}
1. & tc & = & a \star fz & 7. & tl & = & b \star fz & 13. & sx & = & rx + tg \\
2. & tj & = & b \star fx & 8. & ta & = & a \star fx & 14. & th & = & tb + te \\
3. & tf & = & d \star vz & 9. & td & = & d \star vx & 15. & wy & = & vy + tk \\
4. & tk & = & b \star fy & 10. & wz & = & vz + tl & 16. & ti & = & tc + tf \\
5. & tb & = & a \star fy & 11. & tg & = & ta + td & 17. & sz & = & rz + ti \\
6. & te & = & d \star vy & 12. & wx & = & vx + tj & 18. & sy & = & ry + th
\end{array}$$

The problem is to refactor P1 into objects. Any physicist can do that in seconds, particularly if told what the variables mean. But the point is what SCA can do using P1 as the *only* Physics it knows. Thus, we assume that P1 is a thought experiment and its statements are measured "laws" of Physics.

The numbers in P1 are primary keys for the services, and, since the canonical form establishes a one-one correspondence between services and domains, also for the domains and their respective variables [2, Section 3]. The correspondence also simplifies notation, since the entire matrix can be specified by giving the permutation of the primary keys. The input variables are $a, b, d, fx, fy, fz, rx, ry, rz, vx, vy, vz$, numbered 19-30. Input variables do not participate in the analysis because the algorithm can not control when and where they are calculated. The output variables are sx, sy, sz, wx, wy, wz. They do participate in the analysis but with zero scope, because the algorithm can not control their scope. They are not sources of data flows.

The canonical matrix C_1 corresponding to P1 is given in Figure 1. The input variables have been added to the right of C_1 for reference. C_1 has been obtained by transformation of the program, followed by a permutation of the columns as necessary to bring the C's to the diagonal. The matrix of sequences Q is not used for linear sections because the sequence of the services is the order of the rows. C_1 is completely specified by the permutation (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18). The profile of C_1 is 81, the maximum width of the data channel is 15, and the average width is 4.50. The matrix does not show any clearly defined patterns, or any feasible partitioning that would result in empty submatrices. Therefore its block sparsity is 0.

	OP	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	*	C																		A					A						
2	*		C																		A		A								
3	*			C																		A									A
4	*				C																A			A							
5	*					C														A				A							
6	*						C															A								A	
7	*							C													A				A						
8	*								C											A			A								
9	*									C												A							A		
10	+							A			C																				A
11	+								A	A		C																			
12	+		A										C																A		
13	+											A		C												A					
14	+					A	A								C																
15	+				A											C														A	
16	+	A		A													C														
17	+																A	C										A			
18	+														A				C								A				

Figure 1: Matrix C_1 for Program P1 in canonical form, with the input variables 19-30 added to the right. This matrix has a profile of 81, the maximum width of the data channel is 15, and the average width is 4.50. The indicated numbers are primary keys for the service-domain pairs, not row or column indices.

		OP	3	1	16	17	4	15	6	5	14	18	2	12	9	8	11	13	7	10	19	20	21	22	23	24	25	26	27	28	29	30
	3	*	C																				A									A
$S1$	1	*		C																	A					A						
	16	+	A	A	C																											
	17	+			A	C																							A			
$T1$	4	*					C															A			A							
	15	+					A	C																							A	
	6	*							C														A								A	
$S2$	5	*								C											A				A							
	14	+							A	A	C																					
	18	+									A	C																A				
$T2$	2	*											C									A		A								
	12	+											A	C																A		
	9	*													C								A							A		
$S3$	8	*														C					A			A								
	11	+													A	A	C															
	13	+															A	C									A					
$T3$	7	*																	C			A				A						
	10	+																	A	C												A

Figure 2: Matrix C_2 for Program P1, obtained from matrix C_1 of Figure 1, by applying the SCA algorithm. The profile of C_2 is 15, the maximum width of the data channel is 2, and the average width is 0.83. The corresponding values for C_1 are 81, 15 and 4.50. C_2 is partitioned into 36 submatrices, 30 of which are empty, resulting in a block sparsity of 83%. The 6 non-empty submatrices are canonical and identify 2 classes with 3 objects each. Matrix C_2 is SCA2-locked.

The next step is to apply either SCA1 or SCA2 to C_1. In either case, the result is matrix C_2, defined by the permutation (3, 1, 16, 17, 4, 15, 6, 5, 14, 18, 2, 12, 9, 8, 11, 13, 7, 10) and shown in Figure 2. C_2 is canonical and SCA2-locked, meaning that the metrics can not be further improved, at least in the first order. The profile of C_2 is 15, the maximum width of the data channel is 2, and the average width is 0.83. The data channel is now very narrow and well defined.

Matrix C_2 is now ready for being partitioned into objects. The goal of partitioning is to obtain a symmetric block-diagonal form, with populated submatrices only on the diagonal and as many empty off-diagonal blocks as possible. The same partition must be applied to rows and columns. The presence of objects is indicated by narrows in the data channel. In this case, we notice that no scopes cross the boundaries between rows $17-4, 15-6, 18-2, 12-9$, and $13-7$. These boundaries, marked in Figure 2, define the partition with a total of 36 submatrices and a block sparsity of 83 %. The narrows are not the only clues provided by SCA. Linear patterns can frequently be observed. In this case, the column of operators contains the pattern (*, *, +, +, *, +), repeated 3 times, and the scope lengths exhibit the pattern (2, 1, 1, 0, 1, 0), also repeated 3 times, suggesting a partition into nine 6×6 blocks. However, such a partition would have a block sparsity of only 67%.

The final step is to separate the classes into MMC submodels and repeat the procedure to create further objects of a higher level out of the objects just found. The partitioning indicates the presence of 2 classes, say class S of 4×4 and class T of 2×2, with 3 objects each, say S1, S2, S3 and T1, T2, T3, as indicated in the figure. Class S can be designed to have two attributes initialized in the constructor, and one method with one argument. one local variable, and a return value. Class T has one attribute and one method with one argument and a return value. Separating S and T as submodels, matrix C_2 is reduced to a 6×6 canonical matrix, and a repetition of the procedure results in class U derived from S and T with the objects paired as (S1, T3), (S2, T1), and (S3, T2). Finally, class V is found for the input data, with the object associations (fx, rx, vx, sx, wx), (fy, ry, vy, sy, wy), and (fz, rz, vz, sz, wz) with corresponding temporaries (ta, td, tg, tj), (tb, te, th, tk), and (tc, tf, ti, tl), and a, b, d as common input arguments. Classes U and V represent the *Law of Independence of the Components of Motion*, a fundamental symmetry present in the equations represented by P1 and detected by the SCA algorithms. We recall that SCA did not know what the variables "meant". Now it does.

The reader may wonder why SCA1 did not discover vectors. The reason is that nothing in P1 tells SCA about that symmetry. If some indication of its presence is included, for example Euler's equations for the rotation of a rigid body, or a rotation of coordinates that mixes the coordinates but leaves the equations invariant in form, then vectors take precedence over components of motion and SCA1 finds them. In the present case, SCA1 does not detect vectors at all. SCA2, if randomized, detects the permutation (2, 4, 7, 12, 15, 10, 8, 5, 1, 9, 6, 3, 11, 14, 16, 13, 18, 17), which defines vectors. It has a profile of 45 and a data channel with a maximum width of 6 and an average width of 2.50, and can be block-partitioned into 36 submatrices with a block sparsity of 72%. These measures are not very good. The permutation is not SCA2-locked, because (7, 12) and (3, 11) do commute and commuting them would improve the profile. SCA2 continues running and eventually finds the same result as before.

We note that the procedures in this Section can be automated. They are explained here in detail to illustrate how they apply to the MMC. The example is very simple, of course, but we do believe that the SCA algorithms can be used to search thought experiments for fundamental symmetries in the theories of Physics.

3 Conversion from business rules to MMC and executable code

Business rules are quantitative statements that specify the operations of an organization. There is only a small semantic gap between business rules and their corresponding MMC representations. This is also true of technical material such as operation manuals and "how to" manuals. All of them have many features in common with the MMC, making conversion to the MMC frequently easy. This is particularly so in the case of businesses, where parts of the conversion can be automated based on the fact that the MMC is a relational database and many businesses already use relational databases. Furthermore, since the MMC repressentation is a program, it can be compiled, resulting in a direct path from business rules to executable code without using any programming language. It is possible for a business to keep a current MMC model of its operations. The model can aid for testing the effects of strategic decisions before they are implemented. If implemented, changes to the model can be made and compiled directly into a new program.

The existence of the MMC model is guaranteed by the Turing completeness of the MMC. Therefore, in this Section, we only address some practical aspects of the conversion by means of a small example where simple rules that govern a business decision are converted to their equivalent MMC model. A more detailed analysis of this model is carried in Section 4.

Business rules example

The problem statement is: calculate the cost to outfit an aircraft model A, B or C to carry mail (M) or spray crops (S). A simple high-level analysis of this problem indicates that the following services and control variables are needed:

- services AM, BM, CM determine the list of mail equipment for each of model A, B, or C.
- services AS, BS, CS determine the list of spray equipment for each of model A, B, or C.
- services MCost, SCost calculate the cost of mail or spray equipment.
- the control variables are: a (select M/S) and b (select A/B/C).

where all domains are assumed to be unrestricted. The execution algorithm should initialize a and b, call one of AM, BM, CM, AS, BS, or CS to determine the list of equipment, and call one of MCost or SCost to determine the cost, passing the list. This information is enough to create the top level of the MMC transform, containing the *declarations* of the 9 services. The result is shown in Figure 3. To continue the procedure, business rules should be provided and declarative submodels obtained for each of the services, possibly expressed again in terms of additional, more detailed services, and the corresponding sequencing logic. The end result would be a hierarchy of submodels of progressively finer granularities, all the way down to the services provided by the operating system. The final model can be compiled by a modified compiler into executable code.

$$
\mathsf{C} = \begin{array}{|lllllll|}
O & a & b & \text{list} & \text{cost} & a_0 & b_0 \\
\texttt{init} & c1 & c2 & & & a3 & a4 \\
\texttt{AM} & & & c1 & & & \\
\texttt{BM} & & & c1 & & & \\
\texttt{CM} & & & c1 & & & \\
\texttt{AS} & & & c1 & & & \\
\texttt{BS} & & & c1 & & & \\
\texttt{CS} & & & c1 & & & \\
\texttt{MCost} & & & a1 & c2 & & \\
\texttt{SCost} & & & a1 & c2 & &
\end{array}
\qquad
\mathsf{Q} = \begin{array}{l|llll|l}
A & P & a & b & F & \\
1 & & & & \texttt{init} & \\
\hdashline
 & \texttt{init} & M & A & \texttt{AM} & \\
 & \texttt{init} & M & B & \texttt{BM} & \\
 & \texttt{init} & M & C & \texttt{CM} & \\
 & \texttt{init} & S & A & \texttt{AS} & \text{R} \\
 & \texttt{init} & S & B & \texttt{BS} & \\
 & \texttt{init} & S & C & \texttt{CS} & \\
\hdashline
 & \texttt{AM} & & & \texttt{MCost} & \\
 & \texttt{BM} & & & \texttt{MCost} & \\
 & \texttt{CM} & & & \texttt{MCost} & \\
 & \texttt{AS} & & & \texttt{SCost} & \\
 & \texttt{BS} & & & \texttt{SCost} & \\
 & \texttt{CS} & & & \texttt{SCost} & \\
 & \texttt{MCost} & & & \texttt{exit} & \\
 & \texttt{SCost} & & & \texttt{exit} &
\end{array}
$$

Figure 3: The relational model for Section 3, obtained directly from business rules. A relation R of degree 4 with six tuples is marked in matrix Q.

4 Conversions between MMC models and UML class models

In this Section we reuse the top level MMC model of Figure 3 to discuss its conversion to UML class models. We also use it to illustrate important formal analytical capabilities of the MMC, not traditionally associated with UML models. Details of the MMC submodels are not needed to carry out the analysis. Our discussion centers around the implementation of the control variables in the matrix of sequences Q, and the MMC support for inheritance and polymorphism. In addition, in this case, a source of confusion that has remained unnoticed in the literature, is revealed.

UML class models

For the sake of this example, we will produce designs that use combinations of conditional logic and single-inherited polymorphism (SIP). SIP makes decisions based on a single condition at a time, and the designs differ depending on which decision is made first. Let R be the relation of degree 4 indicated in matrix Q of Figure 3. We must transform R into two relations that use a single control variable each. The relational operation that achieves that goal is normalization. R is unnormalized in both a and b, and there are two ways to normalize it.

We start by normalizing domain a first. We note that $a.\nu = \{M, S\}$, where $a.\nu$ is the value of variable a, and augment R with the new domain $\alpha = \{1, 2\}$ containing foreign keys into a. Then, we project R on $\{P, A, \alpha\}$ and on $\{\alpha, b, F\}$, respectively. The result are relations R1 and R2, which have been substituted for R and shown in Figure 4 as part of the normalized matrix Q_{12}. R2 can still be normalized for b, but this will not be necessary. R1 and R2 contain the same information as R. In fact, a join of R1 and R2 on R1.F = R2.P yields R. The new keys in domain α are in fact two new fictitious "do-nothing" services introduced for convenience.

To normalize in the reverse order, starting with $b.\nu = \{A, B, C\}$, we add the fictitious services $\beta = \{3, 4, 5\}$ as keys for domain b, and project R on $\{P, b, \beta\}$ and $\{\beta, a, F\}$, respectively. The remaining normalization with respect to a is superfluous. The result are relations R3 and R4, which we have inserted into matrix Q_{34} and are identified in Figure 4. A join of R3 and R4 on R3.F = R4.P yields R, as it should.

$$
\begin{array}{r}
\\
\mathsf{Q}_{12} =
\end{array}
\begin{array}{ccccc|c}
A & P & a & b & F & \\
1 & & & & \mathtt{init} & \\
\hdashline
 & \mathtt{init} & M & & \mathtt{1} & \\
 & \mathtt{init} & S & & \mathtt{2} & \mathrm{R1} \\
\hdashline
 & \mathtt{1} & & A & \mathtt{AM} & \\
 & \mathtt{1} & & B & \mathtt{BM} & \\
 & \mathtt{1} & & C & \mathtt{CM} & \mathrm{R2} \\
 & \mathtt{2} & & A & \mathtt{AS} & \\
 & \mathtt{2} & & B & \mathtt{BS} & \\
 & \mathtt{2} & & C & \mathtt{CS} & \\
\hdashline
\end{array}
\qquad
\mathsf{Q}_{34} =
\begin{array}{ccccc|c}
A & P & a & b & F & \\
1 & & & & \mathtt{init} & \\
\hdashline
 & \mathtt{init} & & A & \mathtt{3} & \\
 & \mathtt{init} & & B & \mathtt{4} & \mathrm{R3} \\
 & \mathtt{init} & & C & \mathtt{5} & \\
\hdashline
 & \mathtt{3} & M & & \mathtt{AM} & \\
 & \mathtt{3} & S & & \mathtt{AS} & \\
 & \mathtt{4} & M & & \mathtt{BM} & \mathrm{R4} \\
 & \mathtt{4} & S & & \mathtt{BS} & \\
 & \mathtt{5} & M & & \mathtt{CM} & \\
 & \mathtt{5} & S & & \mathtt{CS} & \\
\hdashline
\end{array}
$$

Figure 4: The relevant parts of two different versions of matrix Q, prepared for SIP or conditional implementations. Relations R1, R2, R3 and R4, all of degree 3, are identified.

Matrix Q_{12} involves 3 different decisions, one based on values of a and two based on values of b. If combinations of conditionals (C) and polymorphism (P) are used, then there are 8 possible combinations: CCC, CCP, CPC, CPP, PCC, PCP, PPC, PPP. In the case of Q_{34}, the b-decision is made first, and there are 3 different a-decisions, resulting in 16 different combinations. In total, there are 24 possibilities. Those combinations represent different refactorings of the same OO program. We show a few of them and illustrate the major effect they have on the designs. Even if unlikely combinations such as CCP or CPC are discarded, 8 choices still remain. In a large program, where many conditionals are involved in each one of numerous logical decisions, this can lead to a very large number of choices. In contrast, if MIP is prescribed, there is only one choice.

The first design, shown in Figure 5(a), is similar to the running example in the Survey [8]. It corresponds to matrix Q_{12}, case CPP. The services are provided by class Document. The caller instantiates one of the Document-derived objects (equivalent to initializing b), and calls its Cost method, passing a. Cost selects between MCost and SCost (the a-decision), then invokes the corresponding polymorphic version (the b-decision), which executes the correct helper code to determine the list of equipment, instantiates a Mail or Spray object, and calls its MCost or SCost method for the final cost calculation, passing itself to make the list available to the method.

The second design is shown in Figure 5(b). Here, the caller instantiates one of the Document-derived objects (equivalent to initializing b), and calls MCost or SCost, which create either a Mail or a Spray object (equivalent to initializing a). Then, they call the generic accept method (the b-decision), passing the Visitor object just created, and accept invokes the corresponding polymorphic version of visit* (the a-decision). This design corresponds to matrix Q_{34}, case PPPP, because the decisions are made in the order ba and all four are polymorphic. It is

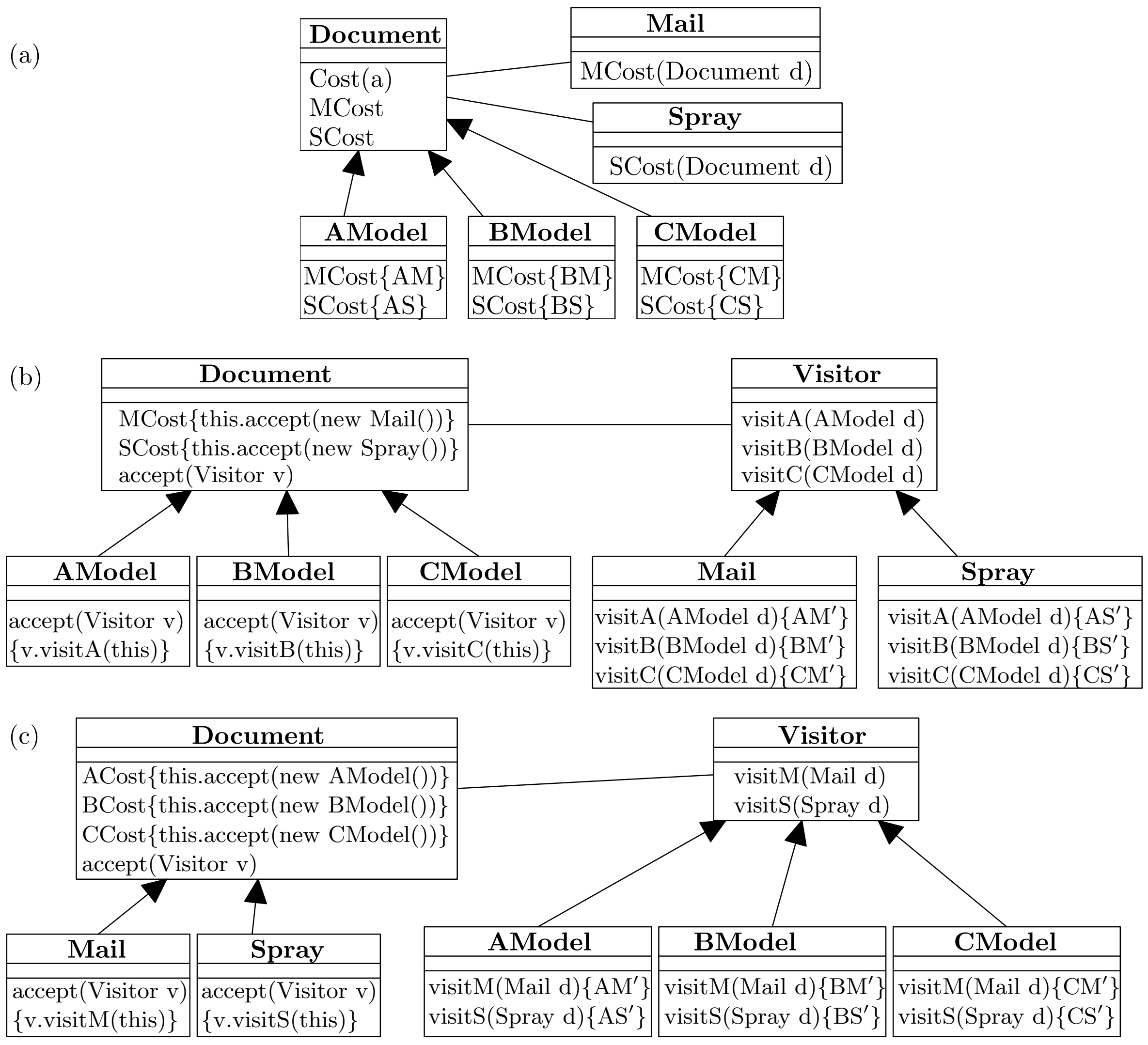

Figure 5: Designs corresponding to: (a) case CPP of matrix Q_{12}, (b) matrix Q_{34}, case PPPP, and (c) matrix Q_{12}, case PPP.

important to note that a *Visitor* design pattern [9] was used in the Survey [8] to obtain a similar design, by refactoring the running example, but the transformation took 20 Fowler refactorings (14, not counting the 6 renames), all at the level of the OO code. This is to be compared with one relational operation on matrix Q_{34} in the MMC case.

The last design, shown in Figure 5(c), is similar to the design of Figure 5(b), but with the roles of model and task reversed. It is based on matrix Q_{12}, case PPP, because the decisions are made in the order *ab* and the *a*-decision and both *b*-decisions are polymorphic.

All three designs have flaws and are difficult to understand. The design of Figure 5(a) has mail and spray functionality in all Document subclasses. An upgrade, say for passenger or cargo transportation, would require changes in all the subclasses, plus the addition of a new Passenger or Cargo class, making it even more complex and less understandable. However, upgrading to a new aircraft model D requires the addition of only one subclass. The design of Figure 5(b) has aircraft model functionality in all Visitor subclasses. An upgrade to model D would require changes in all subclasses, plus the addition of a new subclass to Document. Similar considerations apply to the design of Figure 5(c), where it is now easy to upgrade to model D, but difficult to upgrade to passenger or cargo. The use of patterns, as erroneously advocated in the literature, does not fix the problems, but merely shifts them to another area. The source of the problems is the destruction of the inherent fundamental symmetry between a and b originally present. When SIP was chosen for implementation, we had to establish an arbitrary order between a and b by using matrices Q_{12} or Q_{34}, and symmetry was lost.

5 Conclusions and outlook

It was previously said [2] that every finitely realizable physical system can be perfectly represented by an MMC. This guarantees the existence of the MMC, but questions remained regarding its practicality. This work attempted to shed some light on that issue, and on the breadth of possible applications of the MMC. After considering three very different applications and having laid out the basic ideas for various types of MMC analysis, we have seen encouraging results and gained enough practical experience to believe that the MMC shows substantial promise. It is now possible to recommend further research in several areas as follows.

In Software Engineering: (1) Start with a medium-size human-made OO program, for example in Java, or in C++ with all its preprocessing directives. (2) Transform it to MMC format down to full detail. Existing theorems prove that the representation is possible and perfect. Based on well-known relational database design techniques, and since the MMC is a relational database, this step should be easy. (3) Apply SCA to create classes, inheritance trees, methods. (4) Convert back to the original language, either manually or using a suitable utility, and compare with the original source. Such an experiment should not be very difficult to perform. We would learn a great deal about the MMC, and gain valuable insight about objects as well. We trust the results will compare favorably with human work, and very important consequences will ensue. We recall that in an MMC-centric software environment [1] the MMC representation *is* the program. Executables are compiled from the MMC, tests, verification and certification are performed on the MMC, UML and MDA are obtained from the MMC. The language serves only for man-program communication.

Pilot tests are needed in business. The idea is that a business should keep a current MMC model as its operational repository. To test this concept: (1) Start with a medium-sized business. (2) Transform all its rules to MMC format. (3) Compare MMC output with existing software, databases and operations. (4) Use the MMC to test new strategic business decisions, planned expansions, or user and business software developments. (5) Compare MMC decisions with actual decisions made by managers.

It is said that Physics is 1% inspiration and 99% transpiration. The MMC will impact the transpiration part. In brief: (1) Consider a theory with a relatively simple structure, such as Special Relativity or Matrix Mechanics. (2) Define a set of thought experiments with a physical content equivalent to that of the theory. (3) Cast the experiments into MMC format. The Theorem of Universality guarantees that this is possible. (4) Submodel services present in the high level MMC to full detail. This is also possible for the same reason. Experience with Symbolic Algebra programs may help in the process. An alternative would be to ask Wikipedia authors to produce preliminary MMC transforms of their wikis. (5) Apply SCA and compare the resulting objects with the objects known to physicists.

A fundamental goal of all projects should be to produce preliminary but compatible MMC modules, which can later be expanded and combined to serve the purposes of each particular application. All projects are large and challenging, no doubt, but they are groundwork and will seed future full scale research.

6 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.
[2] Sergio Pissanetzky. 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.
[3] Mario Bunge. “Treatise on Basic Philosophy.” Boston, Riedel, 1977 and 1979.
[4] Shyam R. Chidamber and Chris F. Kemerer. “A metrics suite for object oriented design.” IEEE Trans. Software Engineering, vol. 20, no. 6, pp. 476-493, 1994.
[5] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* Ph.D. thesis, Dep. of Computer Science,

Univ. of Illinois at Urbana-Champaign. 1992.
[6] Alejandra Garrido and Ralph Johnson. "Challenges of refactoring C programs." Proc. International Workshop on Principles of Software Evolution. pp. 6-14, 2002. Orlando, Florida.
[7] Leon Y. Chan, Sriram Kosuri, and Drew Endy. "Refactoring bacteriophage T7." Molecular Systems Biology, article number: 2005.0018. Published online: 13 September 2005.
[8] Tom Mens and Tom Tourwé. "A Survey of Software Refactoring." IEEE Transactions on Software Engineering, Vol. 30, pp. 126–139, February 2004.
[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Languages and Systems." Addison Wesley, 1994.