

# A New Universal Model of Computation and its Contribution to Learning, Intelligence, Parallelism, Ontologies, Refactoring, and the Sharing of Resources

Sergio Pissanetzky, *Member, IEEE, AAAI*

**Abstract** – The recently introduced Matrix Model of Computation in its imperative form (iMMC) can perfectly represent any finite physical system. A new canonical form (cMMC) is introduced. The iMMC is sequential, highly structured, and object-oriented, and saves resources by reusing them heavily. Software and other models convert easily to iMMC and viceversa. The cMMC, instead, minimizes resource reuse, is massively parallel and self-organizing, can refactor a system and find its ontology, and supports learning and AI of indefinite extent or detail. Algorithms and procedures for transformations between the two forms in both directions are included, and proposed as the basis for a unifying theory on the analysis and synthesis of systems. An abstract machine and an extensive case study are also included.

**Keywords** – Artificial intelligence, knowledge representation, learning, object-oriented, ontology, parallel, refactoring.

## I. INTRODUCTION AND PREVIOUS WORK

The Matrix Model of Computation (MMC) in its imperative form (iMMC) is a 2-tuple of sparse matrices, the *matrix of services*  $C$  that contains mappings or *services* and the *variables* that intervene in the mappings, and the *matrix of sequences*  $Q$  that contains the *sequences of execution* for the services.

Several publications describe MMC theory. The iMMC was first introduced as a formal, Turing-complete *virtual machine*, a relational database, and a computer program that can be compiled and executed, all three at the same time, intended for the automation of refactoring and other transformations of computer software [1]. The iMMC has a rich algebra of matrix, algebraic, graph, and relational operations that can be used to design automatic tools for transformations. Any transformation of an iMMC model of source code induces a corresponding transformation on the code. An iMMC-centric software development environment was proposed, which would support the mathematical manipulation of computer programs.

The iMMC was later revisited, but this time as a general, unifying model for systems [2]. The Universal Quantum Computer (UQC) is an abstract machine that can perfectly represent any finite physical system, including any Turing machine. The UQC is more general than the Turing machine because it can also represent quantum systems that can not be represented by any Turing machine. The iMMC was proved to be UQC-complete, giving rise to the following theorem:

**Theorem 1.** *Every finitely realizable physical system can be perfectly represented by a Matrix Model of Computation in its imperative form.*

Consequently, the iMMC was proposed as a universal model of computation. Also introduced in the same publication were the notions of *canonical submatrices* and *service commutation*, and the *Scope Constriction algorithm* (SCA). SCA refactors the iMMC model and automatically associates data and behavior into *objects*, in the object-oriented sense, making the iMMC a self-refactoring program that can find a system's ontology.

Applications of the iMMC were considered and illustrated with examples in several very different areas, such as theories of Physics, business rules, and UML class models, including MMC support for refactoring, encapsulation, inheritance, and polymorphism [3]. Also considered were applications of the iMMC to artificial neural networks (ANNs) and to classification problems [4]. Also discussed in this last publication are the notions that every iMMC model has a natural ontology determined by the information stored in the model, and that the iMMC is dynamically stable and the objects in the ontology are in fact its stable attractors. The iMMC was considered as a universal knowledge representation suitable for AI applications [5]. The applications are so diverse because of the universality of the iMMC.

A complete account of MMC theory is included in Section II of this paper.

The *Scope Constriction Algorithm* (SCA) ([2] §3, [4] §4) is a simple heuristic greedy algorithm for the constrained minimization of the *profile* of a sparse matrix. SCA applies to certain *canonical* matrices that sometimes appear in iMMC models as submatrices of matrix  $C$ . The profile depends on the order of the services in the matrix. The algorithm systematically finds *service commutations* that reduce the profile, and effects those that are legal, subject only to the partial order imposed on the set of services by the *predecessor-successor* constraints. The process induces a *partition* of the matrix into dense diagonal blocks, and may eventually leave some elements outside the blocks. The process can also be viewed as one that *refactors* the matrix and “constricts” – minimizes under the stress of constraints – the scopes of all variables. The effect is to reconcile the flow of data and the flow of control in the canonical matrix, and to *encapsulate* data and behavior into highly cohesive, weakly coupled diagonal modules, or *objects*.

The objects are subsequently sorted into *classes*, inheritance and other object-oriented relationships are determined, and the *ontology* of the system described by the canonical submatrix is obtained. The SCA algorithm is extensively discussed in its deterministic and random versions in Section II-F.

The properties of the SCA algorithm are at the root of the present work. Since the properties are so interesting, but the applicability so limited, it is natural to question whether it can be extended by making the canonical submatrices bigger or more abundant, or even converting an entire iMMC into a canonical form. As discussed in this paper, the answer to both questions is yes. Any general iMMC model can be converted into canonical form partially or entirely by a series of refactoring transformations that profoundly alter its structure but do not affect its overall behavior. This is a process of *system analysis*, and its result is the *canonical form* of the MMC (cMMC). The inverse transformation, or *system synthesis*, is also possible, and SCA is its foundational part. System synthesis minimizes the required resources by reusing them in a way that does not conflict with the system's intended purpose, such as to exhibit parallelism or the ability to learn and develop intelligence. Object-oriented notions are of the essence for synthesis.

Key to the design of the transformations is the notion of *resource reuse*. The resources in the iMMC are the services and the variables. Services are reused when they are invoked more than once or used to represent more than one mapping. Variables are reused when they are assigned values more than once. Reusing any resources forces an artificial *order* on the set of resources, which may or may not be compatible with the features that the system is expected to have. The transformations are designed around the requirement that any superfluous total or partial order must be eliminated, until only the partial order of the predecessor-successor constraints remains. The MMC appears to be the only known knowledge representation capable of supporting such a concept. The cMMC and the transformations are introduced in Section III.

Services in the cMMC are autonomous. There is no controlling order, they execute on their own when their input becomes available, resulting in *massive interconnectivity* and *total parallelism*, both of which are properties of the cMMC. Another property of the cMMC is the *act of learning*, a very simple transformation, the result of which is a cMMC model that includes the new knowledge but is still a cMMC with the same mathematical properties as before and therefore capable of another act of learning. Service autonomy also supports *aggregate learning*, where modules of knowledge are combined by means of multiple acts of learning to form a larger module. The cMMC keeps knowledge and behavior together. It learns by making new connections, and the connections are the knowledge. It grows and adjusts its behavior as it learns, and its ability to learn also grows as it learns, with only practical limits. Many of these properties are strongly reminiscent of the brain, even though the cMMC is a general model of computation, not a specific model of the brain. The basics of MMC supervised learning are presented in Section IV, where a cMMC-based abstract machine that can learn and can be built, is also introduced. Section V covers the case

study, including the teacher instructions, the act of learning, the object-oriented analysis, the advent of a form of intelligence, and issues of implementation. The conclusions are discussed in Section VI, and the Appendix provides a background on directed graphs specifically prepared for this paper.

## II. OVERVIEW OF THE MATRIX MODEL OF COMPUTATION IN ITS IMPERATIVE FORM

The Matrix Model of Computation (MMC) [1] in its *imperative* form (iMMC) is defined as a tuple of sparse matrices

$$\mathcal{M} = (C, Q) \quad (1)$$

where  $C$  is the *matrix of services* and  $Q$  is the *matrix of sequences*. Each matrix describes a set of relations of various degrees. A *relation* of degree  $k$  is the property that assigns truth values to the  $k$ -tuples in the cartesian product of  $k$  domains  $D_1 \times \dots \times D_k$ , not necessarily distinct. The true tuples are the ones explicitly listed in the matrices. Both matrices are *sparse*. A sparse matrix [6] is different from a regular matrix only because a very large fraction of its elements are zero (or null, or blank) and efficient storage schemes and algorithms exist to exploit the sparseness. Sparse matrices can be very large. In the early 80's, a matrix of  $6,000,000 \times 6,000,000$  was reported. Nowadays, the only limitation is the size of the computer.

Both set notation and subscripts are used in this paper. For example, a service  $s$  that appears in row  $i$  of matrix  $C$  and is also a member of a set  $S$  of services may be referred to either as  $s_i$  or as  $s \in S$ , or even as  $s_i \in S$ , depending on context. The terms *set*, *type* and *domain* mean approximately the same, a collection of things, and are sometimes used interchangeably, but are explained more precisely in Section II-A.2. The terms *data* and *information* are also used interchangeably and with the same meaning, as well as the terms *capsule* and *block*. But a distinction is made between the more elementary description of a function  $y = f(x)$ , where  $x$  and  $y$  are *variables* that can "have a value" or "change values" or be "calculated" or "initialized", and the view more common in mathematical logic or set theory that the function is a relation over the domains  $X = \{x\}$  and  $Y = \{y\}$  and the notion of variable does not exist. In fact, the iMMC is defined first in terms of the elementary descriptions, but in the course of the paper the notion of variable is progressively dismissed and the relational descriptions become predominant and essential in the conclusions. Theorem 1 applies to the iMMC. This section contains an extensive account of many of the properties, algorithms and associated mathematics of the iMMC.

### A. The Matrix of Services $C$

The matrix of services  $C$  is defined in terms of a set of *services*  $S$  and a set of *variables*  $V$ . If  $n = |S|$  and  $m = |V|$ , the size of  $C$  is  $n \times m$ , and there is a one-one correspondence or bijective mapping between the services and the rows, and between the variables and the columns. Rows are always ordered and numbered from 1 to  $n$ , and columns from 1 to  $m$ . The order of services and variables, instead, is arbitrary and can be changed with no consequence. The term *permutation*

refers to a rearrangement of the respective bijections, not to a reordering or renumbering of the rows or columns.

A variable  $v_j \in V$  has a *domain*  $D_j$  associated with it, where  $D_j$  is the set of values permitted for  $v_j$ . The value of  $v_j$  can be any element in  $D_j$  and only an element in  $D_j$ . If  $\mathcal{D}$  is the set of all domains, then  $|\mathcal{D}| \leq |V|$ , because there can be more than one variable with the same domain. A variable can have two states, *initialized*, or *un-initialized*. The variable is un-initialized when it has been declared but not yet assigned a value. Once a value is assigned to a variable, the variable is considered initialized. The action of changing the value of a variable is referred to as *re-initialization*. A variable can have other properties as well, such as the *role* it plays in a service. The set of current values of all the variables defines the *state* of the system, and because of that the variables are also called *state variables*. Two states are different if at least one state variable has different values in the two states, otherwise they are the same state. Services and domains are discussed in more detail in the following two sections.

1) *Services*: Informally, a *service* is a procedure that performs some calculations using several variables. A variable is an *argument* if its value is used in the procedure but not modified, a *codomain* if it is assigned a value, or a *mutator* if its current value is used and subsequently changed by the service. The *input* of the service consists of the arguments and mutators, and the *output* of the mutators and codomains. A service is *executed* when new values are calculated for its output variables in terms of its input variables. In general, a service can have its own hidden memory, such as local variables internally defined or constants hardcoded in the procedure, provided the output is always the same when the input is the same. In matrix  $\mathbf{C}$ , an argument is represented with an  $A$ , a mutator with an  $M$ , and a codomain with a  $C$ . In every column there must be at least one  $C$  because every variable must be initialized before it is used, and in every row there must be at least one  $C$  or one  $M$ , because every service must initialize at least one variable, but the remaining elements are optional and can appear in any order. An element in position  $(i, j)$  of matrix  $\mathbf{C}$  corresponds to service  $S_i$  and variable  $v_j$ , and indicates the role played by that variable in the service.

More formally, for any set of variables  $V = \{v_1, v_2, \dots, v_k\}, k > 0$ , and corresponding multiset of domains  $\mathcal{D} = \{D_1, D_2, \dots, D_k\}$ ,<sup>1</sup> where the domains  $D_i$  are not necessarily different, and  $k = |V| = |\mathcal{D}|$ , define  $\widehat{\mathcal{D}}$  as the cartesian product of all the domains in  $\mathcal{D}$ :

$$\widehat{\mathcal{D}} = D_1 \times D_2 \times \dots \times D_k \quad (2)$$

This can also be written as the set of all possible  $k$ -tuples formed by taking one element from each one of the  $n$  domains:

$$\widehat{\mathcal{D}} = \{(d_1, d_2, \dots, d_k) \mid d_i \in D_i, 1 \leq i \leq k\} \quad (3)$$

A  $k$ -tuple in (3) is said to be *marked* or *selected* if all the variables in  $V$  are initialized and their current values are equal to the corresponding values in the tuple, i.e. if  $v_1 = d_1, v_2 = d_2, \dots, v_k = d_k$ . Only one tuple can be selected at a time.

<sup>1</sup>A multiset is a set where any element can appear more than once.

Now, let  $\mathcal{D}(\mathbf{C})$  be the multiset of all domains in matrix  $\mathbf{C}$ , and consider a service  $S$  in some row of  $\mathbf{C}$ . Service  $S$  induces a partition of  $\mathcal{D}(\mathbf{C})$  into the four disjoint subsets  $\mathcal{D}_A, \mathcal{D}_C, \mathcal{D}_M$ , and  $\mathcal{D}_U$ , where the subscripts refer to the domains marked as  $A, C$ , or  $M$  in service  $S$ , or those unused by service  $S$ , respectively. The domains in subsets  $\mathcal{D}_A, \mathcal{D}_C, \mathcal{D}_M$  are the arguments, codomains, and mutators, respectively. Then, service  $S$  is formally defined as the following mapping between cartesian products:

$$S : \widehat{\mathcal{D}}_A \times \widehat{\mathcal{D}}_M \rightarrow \widehat{\mathcal{D}}_M \times \widehat{\mathcal{D}}_C \quad (4)$$

Any of the sets can be empty, except that there must be at least one codomain or one mutator. In terms of the subsets of domains, *execution* of service  $S$  can be viewed as a search operation where one tuple in  $\widehat{\mathcal{D}}_A \times \widehat{\mathcal{D}}_M$  is selected on input and the corresponding tuple in  $\widehat{\mathcal{D}}_M \times \widehat{\mathcal{D}}_C$  is marked as selected on output. Alternatively, one can also say that one tuple in  $\widehat{\mathcal{D}}_A$  and one tuple in  $\widehat{\mathcal{D}}_M$  are selected as input, and one tuple in  $\widehat{\mathcal{D}}_M$  and one tuple in  $\widehat{\mathcal{D}}_C$  are marked as selected on output. In the imperative MMC a service can be executed multiple times, and each time the selected argument tuples and resulting tuple are different. Service  $S$  is said to be *ready for execution* when the following three conditions are satisfied:

- (C1.1) If  $D_A \in \mathcal{D}_A$  is an argument domain in  $S$ , then the variable of  $D_A$  must be in the initialized state.
- (C1.2) If  $D_M \in \mathcal{D}_M$  is a mutator domain in  $S$ , then the variable in  $D_M$  must be in the initialized state and available for re-initialization.
- (C1.3) If  $D_C \in \mathcal{D}_C$  is a codomain in  $S$ , then the variable in  $D_C$  must be either un-initialized and available for initialization, or initialized and available for re-initialization.

Applications of services include functions, orders, links and associations. For example, a total order on a set is a mapping where each element maps to the element immediately following it in the order.

Matrix  $\mathbf{C}$  is frequently constructed by merging services into it. The *merge operation* is as follows. Let  $S$  be the service to be merged. Then: (1) Add a row to  $\mathbf{C}$ , say row  $r$ , and set it in correspondence with service  $S$ . (2) For each codomain, argument or mutator in  $S$ , find the column in  $\mathbf{C}$  with the corresponding variable, say column  $c$ . (3) If such column does not exist, then add one and let that column be column  $c$ . And (4) set the value of element  $C_{rc}$  to  $C, A$ , or  $M$ , respectively.

2) *Variables, domains and domain representations*: Some definitions are necessary in order to make what follows more precise. A *value* is an abstract entity that can participate as an *operand* in an operation, the result of which depends on the value. A set of values, or more precisely the *symbols* that identify them, is a *domain*. A *type* is the set of all values that share a similar functionality. A domain is sometimes defined as a subset of a type. A *function* can be viewed as an *object* with one or more domains as its attributes and the operation as its behavior. To fully define the function a *mapping* is necessary where the results of the operation are defined for all possible values in the domains. A *variable* over a domain simply selects one of the values by marking it as “present” and

the remaining values as “not present.” Passing a variable to a service in matrix  $C$  notifies the service of the presence of the value and allows the service to assume that value’s behavior, as defined by the mapping of (4). When the service sets its output variables, it is again marking certain values in the codomains as present. The mapping that defines the behavior of the values is only known to the service, and the mere presence of the values suffices to complete the operations. The domains do not need to have any knowledge of the functionality associated with the values. A variable can have multiple *lives*. A life begins every time the variable is assigned or reassigned to, even if the new value is the same it already has, and ends with the next reassignment or when service *Exit* is reached. Different lives of a variable can not coexist.

In the MMC, all domains are finite. However, the MMC can still handle infinite domains, either countable or uncountable, in the same way we humans do: by using the finite techniques explained in finite textbooks, based on the concepts of limit, recursion, and iteration, or even some less traditional methods such as harmonic analysis [7]. A function with infinite domains can be handled as long as a finite, recursive algorithm exists that provides access to the mapping. In this case, the infinite domain is described by the finite algorithm, and the algorithm itself by a finite MMC *submodel* [1] where all domains are finite. More advanced concepts, such as the definitions of infinite sequences, derivatives, and integrals, are described by the corresponding algorithms. This resolves a fundamental inconsistency of many programming languages, where some domains are declared as infinite but finite hardware is used to deal with them.

In addition to being finite, and unlike many programming languages, MMC domains are also *tight*. If the set of values that a variable can attain is finite, then this set and exactly this set is the domain for the variable. For example, the domain of variable  $i$  in the C statement `for(int i = 0; i < 3; ++i)` is  $\{0, 1, 2\}$ , not *int*. Conversions of existing software into MMC format should take this important difference into account. The domains can be adjusted by *domain tightening* once transformation SV below has been completed. Since out-of-bound values are a common cause for software failures, domain tightening can add considerably to the safety and reliability of software.

A finite domain such as “aircraft model  $A$ ,  $B$ , or  $C$ ”, is represented by the set of the symbols, in this case  $\{A, B, C\}$ . This is the *direct representation* of domains used in this paper for the iMMC. Other representations are possible. In the *expanded representation*, a variable over a finite domain with  $n$  values is represented by a set of  $n$  boolean variables, all of them over the single domain  $\Delta = \{true, false\}$ , and only one of which can be *true* at a time. Let  $D = \{w\}$  be the domain, where  $w$  represents values, and let  $v$  be a variable over  $D$ . Assume  $v$  is in the initialized state and its current value is  $\bar{w} \in D$ . The extended representation consists of a set  $V = \{\nu\}$  of variables  $\nu$ , all of them over domain  $\Delta$ , and  $|V| = |D|$ . There is a bijective mapping between the values  $w \in D$  and the variables  $\nu \in V$ . To the value  $\bar{w} \in D$  there corresponds the variable  $\bar{\nu} \in V$ . All the variables  $\nu \in V$  are initialized to *false*, except variable  $\bar{\nu}$  which is initialized to

*true*. Conversely, if a variable  $\bar{\nu} \in V$  is initialized to *true*, then the corresponding value  $\bar{w} \in D$  is the current value of  $v$ . The presence of the value  $\bar{w}$  is effectively marked by the *true* value of the variable  $\bar{\nu}$ . This type of representation is used mainly in the canonical form of the MMC.

The *signal-based* representation of domains is very similar to the expanded representation. When it is used for the abstract machine, a change of language is required. A *physical media* – a bus composed of  $n$  wires – plays the role of the  $n$  domains. Each wire has two states, *on* and *off*, so that domain  $\{true, false\}$  becomes domain  $\{on, off\}$ . Only one of the  $n$  wires can be in the *on* state at any time. A service becomes an AND gate, with the argument variables represented by the input signals and the codomain by the output signal. The services are *autonomous*, meaning that they are *driven* by their inputs rather than being caused to execute by the matrix of sequences. The canonical MMC becomes the *circuit* of a parallel computer. Services are never autonomous in the iMMC. The signal-based representation and the autonomy of services are discussed in more detail in Section II-D.2 and III-D.

## B. The Matrix of Sequences Q

The purpose of matrix  $Q$  is to determine the flow of control in the course of an execution run of the iMMC. At the point where a service  $P$  (“previous”) has just finished execution, matrix  $Q$  is responsible for selecting the service  $F$  (“following”) to be executed next. A special service *Exit*, which has no following service, is used to terminate execution and return control to the user.

To make the selection, matrix  $Q$  utilizes a multiset  $\mathcal{K}$  of *control domains*, not necessarily all different, a set  $\mathcal{V}$  of *control variables*, and a set  $\mathcal{Z}$  of *sequences*. Multiset  $\mathcal{K}$  and set  $\mathcal{V}$  are allowed to be empty. There is a bijective mapping between control variables and control domains: every variable in  $\mathcal{V}$  maps to exactly one domain in  $\mathcal{K}$ , and viceversa. A control variable is allowed to assume any value in its corresponding domain. Control variables are state variables and must be initialized by services in matrix  $C$  before being accessed in matrix  $Q$ . There is an injective mapping from variables to columns in matrix  $Q$ : to every variable in  $\mathcal{V}$  there corresponds exactly one column in  $Q$ , and every column has 0 or 1 variables that map to it. Matrix  $Q$  has a total of  $n = |\mathcal{V}| + 2$  columns. Let  $\mathcal{S}$  be the set of all services in matrix  $C$ , let  $\mathcal{P}$  be a domain with the identifiers for all previous services in matrix  $Q$ , which include all services in  $\mathcal{S}$  except *Exit*, and let  $\mathcal{F}$  be the domain of identifiers for all following services in  $Q$ , which include all services in  $\mathcal{S}$ . Then two of the columns in  $Q$  correspond to domains  $\mathcal{P}$  and  $\mathcal{F}$ , respectively, and the remaining  $n - 2$  columns correspond to the control domains. It is customary to represent  $\mathcal{P}$ ’s column first in the matrix, followed with the  $n - 2$  control columns, and  $\mathcal{F}$ ’s column last.

The purpose of a sequence  $Z(P) \in \mathcal{Z}$  is to define a mapping from a previous service  $P$  to one or more possible choices for the following service  $F$ . The selection of  $F$  is based on the *values* – not the roles, as in matrix  $C$  – of 0 or more

control variables. Each possible choice of  $F$  for that particular  $P$  is represented in one row of  $Q$ . Consequently, the sequence  $Z(P)$  may occupy one or more rows. The total number of rows of matrix  $Q$  is  $m = \sum |Z(P)|$ , where the sum extends to all sequences  $Z(P) \in \mathcal{Z}$ . Therefore, the size of  $Q$  is  $m \times n$ .

The definition of a *sequence* is as follows. Let  $P \in \mathcal{P}$  be any service other than *Exit*. To service  $P$  there corresponds exactly one subset  $\mathcal{K}(P) \subseteq \mathcal{K}$  of control domains, not necessarily proper and allowed to be empty. To service  $P$  there also corresponds exactly one sequence  $Z(P) \in \mathcal{Z}$  and one subset of services  $\mathcal{F}(P) \subseteq \mathcal{F}$ . The *degree* of sequence  $Z(P)$  is  $k = |\mathcal{K}(P)|$ , and the tuples in the sequence are all of degree  $k$  as well. Let  $\widehat{\mathcal{K}}(P)$  be the cartesian product of the  $k$  domains in  $\mathcal{K}(P)$ , defined as in (2). Then sequence  $Z(P)$  is formally defined by the following injective mapping:

$$Z(P) : \{P\} \times \widehat{\mathcal{K}}(P) \rightarrow \mathcal{F}(P) \quad (5)$$

Equation (5) simply says that to service  $P$  and to every combination of values of the control variables in  $\mathcal{K}(P)$ , there corresponds a certain following service  $F \in \mathcal{F}(P)$ . At the point where service  $P$  has finished execution, the following conditions must be satisfied:

- (C2.1) If  $K \in \mathcal{K}(P)$  is a control domain used in the sequence  $Z(P)$ , then the variable associated with  $K$  must be in the initialized state and its value must be available.
- (C2.2) If service  $S \in \mathcal{F}(P)$  is a candidate to be executed immediately after  $P$ , then  $S$  must be ready for execution, where “ready for execution” means that  $S$  satisfies conditions C1 of Section II-A.1.

These conditions guarantee that the selection of  $F$  is determined by, and only by the sequence  $Z(P)$ . Or, in other words, all circumstances surrounding the selection of  $F$  to execute after  $P$  have been made explicit via  $Z(P)$ . There are no “hidden agendas”, no dependence on history other than through the control variables, state alone determines the selection of  $F$ . The mathematics that follows, particularly in Section III-C, depends on conditions C2 and would not work without them. An example of a “hidden agenda” would be code inserted by a designer that guarantees that a certain combination of values of the control variables will never happen, and then specifies any arbitrary service  $F$  in correspondence with those values. The procedures of Section III-C can detect such conditions.

The iMMC uses a *routing process* based on a *routing variable*  $r$  to perform tasks such as receiving an initial value for  $P$  from an actor, finding  $Z(P)$ , searching it to determine a value for  $F$ , and invoking service  $F$ . The routing variable  $r$  always contains the value of the identifier of a service in matrix  $C$ , and corresponds to domain  $F$  in matrix  $Q$ , but is also used for searches in domain  $P$ , which is identical to  $F$  except for service *Exit*. The new value of  $P$  needed for the next search after  $F$  has executed, is the same as  $F$ , so there is no need for a separate variable. Variable  $r$  is not usually included among the control variables. Formally, the procedure for execution is as follows:

- (P1.1) An actor sets the input data and an initial value for the routing variable  $r$ , and passes control to the iMMC.

- (P1.2) If  $r$  contains service *Exit*, then stop. Otherwise continue to step P1.3.
- (P1.3) Service  $r$  is invoked for execution.
- (P1.4) A search for  $r$  is performed in domain  $P$ , and the sequence  $Z(P)$  is identified. Conditions C2 are assumed valid, or can be verified as an option.
- (P1.5) Another search is performed in  $Z(P)$  for the (unique) tuple that matches the current values of all the control variables in  $\mathcal{K}(P)$ , and the value of  $r$  is set to the identifier of the service  $F \in \mathcal{F}(P)$  found in that tuple. Continue to Step P1.2.

The *merge operation* for matrix  $Q$  is similar to that for matrix  $C$ . Examples of matrix  $Q$  have been published ([1] p. 145-146, [2] p. 186, and [3] p. 193).

### C. The Control Flow Graph in the iMMC

A directed graph  $G = (V, E)$  called a *Control Flow Graph* (CFG) can be defined for any iMMC model. The vertices in set  $V$  represent all the services in matrix  $C$  and the edges in set  $E$  represent all the tuples in the sequences of matrix  $Q$ . The direction of an edge is taken from service  $P$  to service  $F$  in the corresponding tuple. The edges can be labeled if so desired with the corresponding control variables and the values that enable the link. The vertices can be labeled with the name of the variable initialized by the corresponding service. It is advantageous to regard the CFG as a composition of secluded paths (see Appendix) and convergent and divergent joints. *Joints* are defined as follows:

- A *convergent joint* exists when a vertex has  $n > 1$  incoming edges. In matrix  $Q$ , the identifier of the service at that vertex appears  $n$  times as a destination service in column  $F$ .
- A *divergent joint* exists when a vertex has  $n > 1$  outgoing edges. In matrix  $Q$ , the service at that vertex has a sequence with  $n$  tuples, and control variables are used to select one of them, or, equivalently, to select one of the edges leaving the vertex. The divergent joint is said to be *controlled* by the control variables.

An *execution search* of the CFG is a procedure by which vertices and edges in the CFG are visited in the order determined by execution of the corresponding iMMC. As the search proceeds, vertices and edges are numbered in the order they are found. The search is initiated when an actor passes control to an initial vertex, and then follows the sequence (secluded path – joint – secluded path – joint ...), called the *execution path* or *execution trace*. The search ends when the service *Exit* is reached, or otherwise continues forever. The procedure used by the search is the same as Procedure P1 with the terms vertex and edge substituted for service and tuple, except that the vertices are now numbered. In general, the search may visit any vertex or edge many times, and may end without having visited all the vertices or edges. Secluded paths and convergent joints steer the search in one direction but divergent joints have multiple directions and rely on the control variables to select one. The control variables are in turn determined from input data or by calculations performed by the services. It is

advantageous to consider the nature of these calculations in more detail. The following cases can be identified:

- *Fixed Routing.* Control variables are all initialized from constants and their values do not change during execution. Execution always follows the same path. Divergent joints are sterile, and fronds are either always traversed and result in infinite cycles, or always ignored, preventing the cycles from ever executing.

- *Path-aware Routing.* Some or all control variables are initialized from constants and later recalculated in terms of each other, but not in terms of input data. They will have different values each time execution reaches a divergent joint, and a different route will be chosen each time. Since this effect is independent of input data, the entire execution path can be predicted once and for all. Typical applications include a counter for the number of loops around a cycle, or a detector to determine whether a certain secluded path has been traversed or not.

- *User-controlled Routing.* Some or all control variables depend on the input data. Different sets of input data produce different execution paths. The system can be viewed as an assembly of different, overlapping programs which may or may not have much in common. However, *only one set of input data* can be processed at a time, and *only one execution path* can be followed at a time. The assembly of programs, and the collection of all possible sets of input data, can be partitioned into instances that are path-aware, and each instance set in correspondence with a different time slot. The general case of user-controlled routing can be reduced to multiple instances of path-aware routing.

Only the case of path-aware routing needs to be considered. The execution search can be carried directly on matrix  $Q$ , without recourse to the digraph or the depth-first search. It will most certainly be done that way when applications are developed. However, the digraph approach followed in this paper and the many new concepts and relationships introduced by the depth-first search provide irreplaceable insight, necessary for presentation.

The execution search induces a *total order* on the set of *numbered* vertices and the corresponding edges on the path. Rigorously speaking, the order is only partial for the set of *unnumbered* vertices, because the execution path is not simple, it can overlap or intersect itself and the same vertices and edges can appear more than once in the path. The order becomes total when the set of *numbered vertices* is considered, a set where the elements are the 2-tuples (vertex, number) formed with a vertex and the number assigned to it by the search. This is because a different number is assigned to a vertex each time it appears on the path. The effect of the total order is to eliminate all joints and cycles and convert the entire CFG into a single path, where vertices and edges are organized in the same order that would have been followed by the corresponding services in  $C$  and sequencing tuples in  $Q$  under iMMC execution. The achievement of a total order is a fundamental step towards the conversion of the iMMC to a canonical form, because the canonical form relies on the existence of a total order. The total order is a linear extension

of the partial order imposed by the predecessor-successor constraints. This matter is further discussed in Section III-C.

In the case of path-aware routing, the execution search can be started with an arbitrary set of input data, and amounts to a major *refactoring transformation*. It is a refactoring because it preserves the behavior of the model, and it is a transformation because it results in a new, very different representation of the system. In fact, the execution path is equivalent to the matrix of sequences, and either one of them can be used to describe the system, in addition to matrix  $C$ . Both descriptions have advantages and disadvantages, of course. For example, the matrix of sequences lends itself to the analysis of high level refactoring transformations such as the ones discussed in the literature ([3], p. 192-195). The execution path, instead, provides direct access to the powerful canonical form, suitable for the study of problems such as the automatic generation of ontologies (§II-E).

#### D. Canonical matrices and refactoring transformations

A *canonical path* in the iMMC is a secluded path (see Appendix) such that every service  $S$  on the path satisfies the following 3 conditions:

(C3.1) Service  $S$  has no mutators.

(C3.2) Service  $S$  initializes exactly one variable.

(C3.3) No other service on the path initializes that same variable.

If any services on the path do not satisfy all conditions, transformations exist that can convert them into equivalent services that do. For example, Section III-B has transformations for services that do not satisfy C3.1 or C3.2. To every canonical path in the iMMC there corresponds a *canonical submatrix* of matrix  $C$ , that has very interesting properties. Let  $\mathcal{S}$  be the set of services on the path, let  $\mathcal{V}$  be the set of variables they initialize, and let  $n = |\mathcal{S}| = |\mathcal{V}|$ . There is a one-one bijective correspondence between  $\mathcal{S}$  and  $\mathcal{V}$ . The path establishes a total order on set  $\mathcal{S}$ , in such a way that flow of control can only enter  $\mathcal{S}$  at the first service, follow the services in order, and leave from the last service. The total order applies to set  $\mathcal{V}$  as well, because of the bijection.

The  $n$  services in  $\mathcal{S}$  appear in  $n$  rows of matrix  $C$ , not necessarily consecutive, and the  $n$  variables in  $n$  columns, not necessarily consecutive. However, since the order of rows and columns in matrix  $C$  is arbitrary, it is always possible to permute them in such a way that they become physically consecutive and appear in the order of the path. The intersection of the block of consecutive rows and the block of consecutive columns defines a square  $n \times n$  *canonical submatrix* of  $C$ , say  $\Gamma$ . Matrix  $\Gamma$  is square and lower-triangular, contains no  $M$ 's, all diagonal elements are equal to  $C$ , the upper triangle is empty, and the lower triangle is sparse and contains only  $A$ 's. The upper triangle is empty because no variable can be used in a service before it is initialized, and since every variable is initialized on the diagonal, there can be no  $A$ 's above the diagonal. Once matrix  $\Gamma$  has been identified, matrix  $C$  can be partitioned as follows:

$$C = \begin{vmatrix} C_{11} & \Gamma \\ C_{21} & C_{22} \end{vmatrix} \quad (6)$$

where  $C_{11}$  contains only  $A$ 's,  $C_{21}$  and  $C_{22}$  are sparse submatrices with  $A$ 's,  $C$ 's and  $M$ 's, and  $\Gamma$  is canonical. In Section III, it is demonstrated that any iMMC model can be converted into canonical form in its entirety by applying the transformations discussed in that Section. In such case, matrices  $C_{11}$ ,  $C_{21}$ , and  $C_{22}$  are empty. A small example of a canonical matrix is shown in Fig. 1(b), and another can be found in the literature ([3], Fig. 1).

A service  $S$  in  $\Gamma$  that initializes an argument of another service  $S'$  in  $\Gamma$  is a *predecessor* of  $S'$ , and  $S'$  is a *successor* of  $S$ . A service can have 0 or more predecessors and 0 or more successors. Two services that are not bound by a predecessor-successor relationship, are said to be *commutative*. The *predecessor-successor constraints* establish that the execution of all the predecessors of a service must precede that of the service. The constraints induce a partial order on the set of services in  $\Gamma$ , compatible with, but weaker than the total order of the canonical path.

The fact that commutative services do not participate in the partial order has important practical consequences. Its main effect is to stem the imperative control of the matrix of sequences  $Q$ , which then becomes irrelevant and can be eliminated, as well as the corresponding control variables, leaving matrix  $\Gamma$  alone in full control. A large number of "hidden" degrees of freedom become exposed, and can be used to design new procedures and formal algorithms that manipulate the system mathematically in new, powerful ways. *Service autonomy* (§II-D.2) is one example of the effect of such exposed degrees of freedom. Intelligence is another. Intelligence is an ability, not a state, the ability to adapt, but a system can not adapt if it lacks degrees of freedom.

1) *Service commutation and Proposition 1*: Let  $S_i$  and  $S_{i+1}$  be two commutative services that happen to be in the consecutive rows  $i$  and  $i+1$  of  $\Gamma$ . Then their order of execution is irrelevant and they can be *commuted*. Let  $v$  and  $v'$  be the two variables they initialize. Then  $v$  and  $v'$  are in consecutive columns as well. The operation of *commutation* is a symmetric permutation of  $\Gamma$  that reverses the order of the two rows and that of the two columns. Commutation does not violate the partial order, and the symmetry of the permutation guarantees that  $\Gamma$  remains lower-triangular. Commutation is an *elementary refactoring* of the system, because it is a transformation but it does not affect the behavior.<sup>2</sup>

Repeated commutation may be possible. Let  $n$  be the order of  $\Gamma$ . The  $(S_i, S_{i+1})$  commutation moves service  $S_i$  to row  $i+1$ . Now  $S_i$  and  $S_{i+2}$  are adjacent. If they happen to be commutative, then another commutation can be effected. The procedure can be repeated until  $S_i$  reaches a row, say  $i_B \geq i$ , where either  $i_B = n$  or further commutation with the service in row  $i_B + 1$  would be illegal. Similarly, starting from the original position of  $S_i$  in row  $i$ , it may be possible to commute  $S_i$  with  $S_{i-1}$ , then with  $S_{i-2}$ , and so on, until  $S_i$  reaches a certain row  $i_A \leq i$  where either  $i_A = 1$  or further

commutation with the service in row  $i_A - 1$  would not be legal. The inclusive range of rows  $(i_A, i_B)$ , where  $i_A \leq i_B$ , is the *range of commutation* of service  $S_i$ . Finding the range of commutation for a service amounts to finding the nearest predecessor and the nearest successor of that service.

Fig. 1(b) shows a small example of a canonical matrix. The services in rows 4 and 5 commute because  $\Gamma_{5,4}$  is null. The range of commutation of  $S_4$  is (3, 6). The services in rows 5 and 6 do not commute because  $\Gamma_{6,5}$  is not null. If an attempt were made to commute them, the order of the  $C$  and the  $A$  in column 5 would be reversed and behavior would not be preserved. If services 4 and 5 are commuted, columns 4 and 5 should also be commuted in order to restore the matrix to lower-triangular form. The range of commutation of service  $S_5$  is (4, 5).

The transformation induced in  $\Gamma$  by repeated commutations can be symbolically represented by the following expression [4]:

$$\Gamma' = P^T \Gamma P, \quad (7)$$

where  $P$  is a boolean *permutation matrix*, an  $n \times n$  matrix with only one element equal to *true* in each row and each column, and the remaining elements equal to *false*,  $P^T$  is the transpose of  $P$ , the property  $PP^T = P^T P = I$  holds true, where  $I$  is the unit boolean diagonal matrix, and the rules of matrix multiplication are extended to boolean matrices. The product of two permutation matrices is a permutation matrix, so  $P$  can always be considered as the product of elementary permutations, each involving two adjacent services and the corresponding columns.

Equation (7) says that many different representations of a system are possible, depending on the selection of matrix  $P$ . Consequently, the selection of  $P$  can be viewed as a way to engineer the properties of the systems, and a theory of systems can be established where the properties are studied analytically in terms of  $P$ . The basic proposition for the theory to answer can now be formally posed as follows:

**Proposition 1.** *An important class of problems of system analysis and system engineering can be reduced to the study of permutations of canonical matrices.*

Since the transformation described by (7) is a refactoring, the problem of refactoring is certainly a member of the class. The equation indicates that any system can be refactored, and is the foundation for a *generalized theory of refactoring* that would allow refactoring transformations to be studied analytically, from the inside out, unlike more traditional methods that address refactoring heuristically and from the outside in.

The *Scope Constriction Algorithm* (SCA) (§II-F) is a procedure used for the design of matrix  $P$ , and several SCA-based system design problems discussed in this paper are also members of the class of Proposition 1. They include self-organization, the automation of ontologies, and some forms of learning and AI.

2) *Concurrency, service autonomy, and parallelism*: Commutative services satisfy all three of *Bernstein's conditions* for parallelism, because their outputs are disjoint and the input of each is disjoint from the output of the others. As

<sup>2</sup>The term "refactoring" was originally proposed for object-oriented software [8] but was later extended to a diversity of systems including legal systems [9], RNA [10], the brain and other organs [4], non-OO languages such as C [11], html and sql, hardware, including analog hardware, and even theories of science [3]. In this paper, the term applies to all systems.

a result, canonical matrices possess the important property of *concurrency*. Let  $\Gamma$  be a canonical matrix. Matrix  $\Gamma$  is said to have entered a *concurrency mode*, or simply to be *concurrent*, when two or more commutative services in  $\Gamma$  become simultaneously ready for execution. A service  $S$  is *ready for execution* when it satisfies conditions C1 of §II-A.1. Conditions C1.2 and C1.3 are automatically satisfied in any canonical matrix. Condition C1.1 simply means that, at a certain point in time, all the arguments of  $S$  are in the initialized state. Since commutative services satisfy Bernstein's conditions, when two or more of them become simultaneously ready for execution, they can execute and initialize their codomain variables in any order, or even concurrently. Such a condition may arise, for example, if all the arguments of two commutative services  $S$  and  $S'$  are in the initialized state, except for one, which happens to be the same for  $S$  and  $S'$ . When this last argument gets initialized,  $S$  and  $S'$  become ready for execution at the same time, and  $\Gamma$  enters concurrency mode.

Concurrency lays the foundation for parallelism. Concurrency is a *chain* phenomenon. If a group of services execute concurrently, the variables they initialize become available for other services to use as arguments. This may in turn result in another group of services becoming concurrent and executing simultaneously. The result is the formation of many *cascades* of execution, which are independent and execute in parallel, subject only to the predecessor-successor constraints. Cascades are similar but not identical to *multi-threads*. As an example of concurrency, 6 cascades are possible in the canonical matrix in ([3], §2).

In the iMMC, a service  $S$  executes only when control is passed to it by some process, say  $\Pi$ . Unbeknownst to service  $S$ , process  $\Pi$  has examined the sequences in matrix  $Q$  and has determined that  $S$  is next in line. Service  $S$  is said to be a *slave* of process  $\Pi$ . But process  $\Pi$  also consists of services and is in turn a slave of some other process, and so on, up to the highest level – the user. Such imperative control may stand in the way of applications that expect the system to have a certain degree of autonomy, such as AI and parallelism.

On the other hand, the considerations in this section indicate that condition C1.1 – the requirement that all arguments be in the initialized state – is the *only* requirement for a service in a canonical matrix to be ready for execution. Furthermore, condition C1.1 is a *local* condition, because, unlike the imperative control, it applies to each service individually. As a consequence, it becomes possible to lift the imperative control and to grant *autonomy* to services, based on the following definition:

*An autonomous service is one that can sense its environment, determine when all its input arguments become initialized, and immediately execute on its own and initialize its codomain variable.*

Autonomy puts services in an active role. The requirement that services *sense* their environment and *react* to it decentralizes sequencing control, makes it a local responsibility for the services themselves, allows the strong order in  $\Gamma$  to be removed, and effectively grants autonomy to the services. Matrix  $Q$  can be eliminated, as well as all the control variables

and associated code, resulting in an important simplification. Service autonomy is similar to the well known *dataflow paradigm*, where instructions execute whenever data flows to them.

But autonomous services are more general than it would appear from the simple definition. State variables in the autonomous MMC (aMMC) can be viewed more like *signals*, and services like *devices*, or more specifically like AND gates (see (27)), where the arguments are inputs and codomains the output. Initializing a variable is like setting a signal to ON. The connections *are* the program, and the same connections determine how data flows. The processor is in the gates, because that's all what remains from the services. The "connections" can be actual wires, chemical signals, roads, communication channels, supply lines, wireless signals, variables that carry data in a set of mathematical equations or a computer program, grammatical links or affinities between parts of a sentence or chapters of a book.

The canonical system begins to look like a machine. If the signals are transmitted by wires or dendrites, it looks like hardware of wetware. If not, then the services are like independent *agents* with sensors that can perceive their environment and act upon it by setting their codomain variables. The services become members of a *society* of agents, with which they interact. The *source* of the signals is irrelevant. It is only the presence or absence of a signal what matters to an autonomous service. The service can interact with a constantly changing environment and react to it as long as it senses the right combination of signals. The static, fixed connectivity of the cMMC is replaced in the aMMC with a dynamic scheme where a service can connect with any other service capable of activating its input. The features of autonomy – a local phenomenon, the ability to sense and react to the environment – are very similar to those of intelligence. In fact, I believe that service autonomy and intelligence are one and the same. Intelligence is how autonomy looks to the outside observer.

Service autonomy also means that many services can decide to run at the same time. Parallelism ensues, and it can be of a very high degree if  $\Gamma$  is large. Granting execution autonomy to services also grants them *commutation autonomy*, because commutative services satisfy Bernstein's conditions and can execute in any order or at the same time. Commutation autonomy alone does not result in self-organization. However, if the services compete for some resource, such as energy, then they will tend to self-organize in a way that makes a more efficient use of that resource. In the example shown in Fig. 1 the cascades are  $\{1, 2, 3\} \rightarrow \{4, 5\} \rightarrow 6$ , where the braces indicate sets of services that can run concurrently.

### E. Ontologies and their automatic generation

So far, mostly static properties of  $\Gamma$  have been considered. Attention now turns to more dynamic features, such as transformations, the selection of matrix  $P$  in Equation (7), the flow of data and control in a canonical matrix, and the automatic generation of the natural ontology of a system. Some definitions are necessary for that purpose, and the example in Fig. 1 can help to explain them.

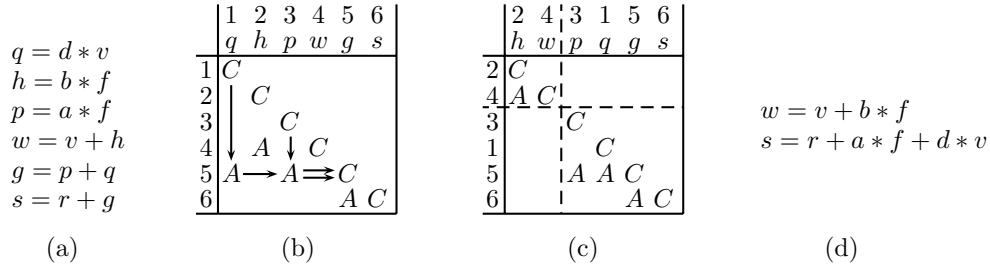


Fig. 1. A running example for Sections II-D – II-F. (a) A fragment of code. (b) The canonical submatrix for the given code. Some data flows are indicated with arrows. Variables  $a, b, d, f, r$  and  $v$  are input data. (c) The canonical submatrix as refactored by SCA. (d) The refactored code, as obtained by a developer.

1) *Scope, profile, lifecycle*: Consider the fragment of code in Fig. 1(a), and the corresponding canonical matrix shown in Fig. 1(b), where the columns have been labeled with the names of the corresponding variables. Variables  $a, b, d, f, r, v$  are input data and have been omitted, and variables  $w$  and  $s$  are output data. If  $\Gamma$  is of size  $n \times n$ , and subscripts  $i, j$  are used to refer to rows and columns, respectively, both in the range 1 to  $n$ , then a variable  $v_j$  in column  $j$  is initialized by the service in row  $i_I(j) = j$  and used as an argument by 0 or more services below row  $i_I(j)$ . Let  $i_T(j)$  be the row of the *last* service that uses  $v_j$ . Then at the intersection of row  $i_T(j)$  and column  $j$  of  $\Gamma$  there is an  $A$ , called the *terminal A* because this is the last time that variable  $v_j$  is used. Variable  $v_j$  is no longer needed below the terminal  $A$ , and can be destroyed at that point. All the services that have a  $C$  or an  $A$  in column  $j$  are said to be *variable-associated* among themselves by way of variable  $v_j$ . Likewise, if  $S$  is a service in some row, all the variables that participate in  $S$  are *service-associated*. Sometimes, they are simply called “the services of variable  $v_j$ ”, or “the variables of service  $S$ .” The extent of column from the initializing  $C$  to the terminal  $A$  is the life of  $v_j$ . The lives of the variables in  $\Gamma$  are entirely contained within  $\Gamma$  itself. Encapsulation has occurred, and it can be said that  $\Gamma$  *owns* the variables and exercises control over them. The *vertical scope* of  $v_j$  is the set of cells in column  $j$  where  $v_j$  stays in scope, including the terminal  $A$  but not the initializing  $C$ , or the empty set if there is no terminal  $A$ . The size of the vertical scope is

$$\sigma_V(v_j) = i_T(j) - i_I(j), \quad (8)$$

where  $i_T(j) = i_I(j)$  if there is no terminal  $A$ . The sum of the sizes of all scopes happens to be the vertical *profile* ([6], p. 15) of  $\Gamma$ :

$$p_V(\Gamma) = \sum_{j=1}^n \sigma_V(v_j) = \sum_{j=1}^n (i_T(j) - i_I(j)). \quad (9)$$

Similarly, let  $i$  be any row of  $\Gamma$ , let  $j_I(i)$  be the column of the left-most  $A$  in row  $i$ , and  $j_T(i) = i \geq j_I(i)$  be the column where the  $C$  is. The horizontal scope for row  $i$  is defined as  $\sigma_H(v_i) = j_T(i) - j_I(i)$ , and the horizontal profile is:

$$p_H(\Gamma) = \sum_{i=1}^n \sigma_H(v_i) = \sum_{i=1}^n (j_T(i) - j_I(i)). \quad (10)$$

In Fig. 1(b), variable  $q$  is initialized in row 1 and the terminal  $A$  is in row 5. The life of  $q$  extends in column 1 from row 1

to row 5, the length of the scope is 4, the vertical profile is 9, and the horizontal profile is 7.

2) *Elementary objects*: The pair formed by a service and the variable it initializes is an *elementary object* or *object element*. In the ontological sense [12], an object is an association between a set of attributes and the behavior based on those attributes. The behavior of a service is the action of initializing the variable, the variable is the attribute, and the other variables used by the service are data necessary to perform the action. The *class* of an elementary object is the domain of the corresponding variable. It can be said that the canonical matrix  $\Gamma$  defines a *primitive ontology*, composed of the elementary objects and their classes. In Fig. 1(b), variable  $q$  and the service in row 1 form an elementary object, that uses input data to initialize variable  $q$ . The domain of  $q$  is the class of the object, and the service itself is the *constructor* of the class.

3) *The flow of information and control*: As execution proceeds in the canonical matrix  $\Gamma$ , the services execute in the order they appear, and the variables get initialized also in the order they appear. Control flows from service to service, and since the  $C$  on the diagonal symbolizes the service/variable association it can be said that the flow of control follows a regular path along the diagonal. As control flows, so does information, but the paths followed by it may be intricate and entangled, and considerably depart from control paths. Information carried by a variable flows vertically from the point on the diagonal where the variable is initialized to the services that use the variable, and then horizontally within the services to the columns where it is used to initialize other variables. This path is called an *information flux line*. The set of flux lines represents an *information channel*, a channel where information flows. The maximum width of the channel is the length of the longest scope, and the average width is  $p(\Gamma)/n$ , where  $p(\Gamma)$  is the profile of  $\Gamma$  given by (9), and  $n$  is the dimension of  $\Gamma$ . Fig. 1(b) shows several flux lines.

Flux lines represent a flow of information across inter-row or inter-column boundaries. Sending a high information flux across many inter-row or inter-column boundaries is undesirable. The *information flux*  $\Phi(i, j)$  crossing from row  $i$  into row  $i + 1$  in column  $j$  is defined as the number of information flux lines passing across that boundary. The only values allowed for  $\Phi(i, j)$  are 0 or 1. The total information flux  $\Phi(j)$  in column  $j$  is the sum of  $\Phi(i, j)$  for all rows. The total information flux  $\Phi(i)$  out of row  $i$  and into row  $i + 1$  for

all columns is the sum of  $\Phi(i, j)$  over all columns. And the total information flux  $\Phi(\Gamma)$  for the entire canonical matrix  $\Gamma$  is the sum of  $\Phi(j)$  for all columns:

$$\begin{aligned}\Phi(j) &= \sum_{i=1}^n \Phi(i, j) \\ \Phi(i) &= \sum_{j=1}^n \Phi(i, j) \\ \Phi(\Gamma) &= \sum_{j=1}^n \Phi(j) = \sum_{j=1}^n \sum_{i=1}^n \Phi(i, j)\end{aligned}\quad (11)$$

and obviously  $\Phi(\Gamma) = \sum_{j=1}^n \Phi(j) = \sum_{i=1}^n \Phi(i)$ . The total information flux satisfies the following important theorem:

**Theorem 2.** *The total information flux in a canonical matrix is equal to the profile of the matrix.*

The theorem is easy to prove. Consider variable  $v_j$  in column  $j$ , and let  $i_I(j)$  be the row where  $v_j$  is initialized, and  $i_T(j)$  the row with the terminal  $A$ . As given by (8), the size of the scope of  $v_j$  is  $\sigma(v_j) = i_T(j) - i_I(j)$ . A flux line flows from row  $i_I$  to row  $i_T$ , and in the process crosses exactly  $i_T - i_I$  inter-row boundaries. Thus  $\Phi(i, j) = 0$  if  $i < i_I(j)$  or  $i \geq i_T(j)$  and  $1 \leq i_I(j) < i_T(j) \leq n$ , and we have:

$$\sum_{i=1}^n \Phi(i, j) = \sum_{i=i_I(j)}^{i_T(j)-1} \Phi(i, j) = i_T(j) - i_I(j) = \sigma(v_j). \quad (12)$$

The total flux for the matrix is obtained by taking the sum of the previous expression for all columns:

$$\Phi(\Gamma) = \sum_{j=1}^n \sum_{i=1}^n \Phi(i, j) = \sum_{j=1}^n \sigma(v_j) = p(\Gamma), \quad (13)$$

where (9) has been used. This proves the theorem. Similar considerations and a similar theorem apply to  $\Psi(j, i)$ , the information flux in row  $i$  from column  $j$  to column  $j + 1$ , except that the value of  $\Psi(j, i)$  can be any non-negative integer. In the example of Fig. 1(b), variable  $q$  receives information in row 1, but the information has to flow along the scope of  $q$  all the way to row 5, where it is received by the service in that row. From there, it flows horizontally to column 5, where it is combined with information transported by variable  $p$  in column 3 and finally used to initialize variable  $g$  in column 5, as the arrows indicate. Information in variable  $h$  flows to row 4 and then into column 4, crossing the path of  $p$ . The information flux is 4 in column 1 and 6 in row 5. The total flux is 3 from row 3 to row 4, and 9 in the entire matrix. The vertical profile is also 9. The data channel is very wide in relation with the size of  $\Gamma$ , it has a maximum width of 4 and an average width of 1.5. The flow of data is said to be *turbulent*.

4) *The reconciliation of flows and Proposition 2:* The preceding considerations suggest that a system's representation can be quite disorganized, even if cast in the form of a canonical matrix. In ordinary computers, the flow of data and the flow of control are separated. But such a separation is the result of analysis, not a primitive of intelligence. The analysis can not be performed without the intelligence, which means

that the two flows must be reconciled first. With this goal in mind, I have proposed [2] the following strategy for the design of matrix  $P$  in (7):

**Proposition 2.** *The flow of information and the flow of control in the canonical matrix are reconciled by systematically reducing the size of the scopes of the variables.*

Proposition 2 is one way to engineer the properties of  $\Gamma'$  in (7). Other choices are possible, such as: improve the performance of a program by *increasing* intermodular coupling or the number of temporary variables; prepare a program for compilation; intentionally cause a separation between data and behavior in order to match the representation to a specific natural or programming language for translation, or to a specific networks; improve the *understandability* of code under development. Proposition 2 is the only one examined in this paper.

The concept relies on the fact that the size of the scope  $\sigma(v_j)$  of a variable  $v_j$ , given by (8), depends on the order of the services, meaning that service commutation can be used to control the scopes and obtain a design for matrix  $P$ . The hope is that the strategy would bring data flows closer to the diagonal, promote modularization and encapsulation, and enhance the organization of the canonical matrix. But the problem of scope minimization is a coupled one, because each commutation affects several scopes at once. Furthermore, because of commutation rules, the problem is also one of constrained optimization, and must be addressed as a whole. A non-negative definite functional or *energy function* is needed, the minimization of which would result in a reduction of scopes in bulk, and, hopefully, in the best possible reconciliation between data and behavior. I have proposed [2] the profile of the canonical matrix, defined in (9), as the energy function to be minimized. Since scope sizes are non-negative numbers, and the profile is the sum of all scope sizes, minimizing the profile should serve the purpose. The equation

$$p(\Gamma') = p(P^T \Gamma P) = \min \quad (14)$$

expresses the mathematical condition for the design of  $P$ , and the Scope Constriction algorithm is the procedure that implements the condition. The process that minimizes the profile also minimizes two other important parameters: the average width of the data channel, and by Theorem 1, the global information flow in the canonical matrix. It is in this sense that the design of matrix  $P$  in (7) by profile minimization implements the strategy put forward by Proposition 2.

#### F. The Scope Constriction algorithm

Two versions of the Scope Constriction algorithm (SCA) were initially proposed [2], and a third version with a better asymptotic complexity was later presented [4]. Details of the algorithm have been covered and are not repeated here, but an overview follows. SCA operates as follows:

- (P2.1) Select an arbitrary unvisited row  $i$  of  $\Gamma$ .
- (P2.2) Determine the range of commutation  $(i_A, i_B)$  for the service in that row.

- (P2.3) For each  $k, i_A \leq k \leq i_B$ , calculate the profile  $p(\Gamma_k)$ , where  $\Gamma_k$  is obtained from  $\Gamma$  by permuting the service from row  $i$  to row  $k$ .
- (P2.4) Select the  $k$  that minimizes the profile, and permute service  $i$  to row  $k$  and the associated variable to column  $k$ .
- (P2.5) Repeat steps (1-4) until all rows are exhausted, and repeat the entire procedure until no more reductions are obtained.

It is important to emphasize that the algorithm is very simple and completely general. SCA applies to any system because it does not rely on any features of the system. SCA only attempts to minimize the profile of  $p(\Gamma)$  of (14) by systematically permuting services and their corresponding variables. The consequences that ensue, however, reach far beyond what one might expect.

1) *Encapsulation*: Let  $v$  be a variable,  $\mathcal{S}(v)$  be the corresponding set of  $v$ -associated services, and  $\mathcal{V}(v)$  be the set of variables initialized by the services in  $\mathcal{S}(v)$ , including  $v$  itself. Then the definition of scope as a set of cells in Section II-E implies that the size  $\sigma(v)$  of the scope of  $v$  can only be reduced by packing the services listed in  $\mathcal{S}(v)$  closer together. When SCA does the necessary permutations, a process of *encapsulation* takes place. The permutations attempt to pack the services into neighboring rows as densely as they can, and they do that by driving all other intervening services away. The process defines a well-differentiated *block of rows*, containing the associated services in set  $\mathcal{S}(v)$ . Because the permutations are symmetric, the process also encapsulates the variables listed in  $\mathcal{V}(v)$  into the corresponding symmetric *block of columns*. The intersection of the block of rows and the block of columns defines a square *capsule*. The remaining services in the matrix form other blocks of rows and columns and other capsules. The blocks of rows and columns define a *partition* of the canonical matrix where the *block submatrices* are the capsules. The meaning of the partition and the procedures used to determine it are discussed in Section II-F.4.

Of course, encapsulation is a complex process. Service commutation constraints and scope couplings cause conflicts. The profile decreases smoothly, but individual scopes behave in a complicated way as they get progressively “constricted” against the constraints. Some of the services in  $\mathcal{S}(v)$  can’t make it into the block of rows, or some of the intervening services are not expelled from it. One particular case is of interest. Let  $v'$  be an intervening variable with a set of associated services properly contained in that of  $v$ :

$$\begin{array}{lcl} v' & \neq & v \\ \mathcal{S}(v') & \subset & \mathcal{S}(v) \end{array} \quad (15)$$

This means that  $v'$  is initialized by a service in  $\mathcal{S}(v)$  and used as an argument only by other services in  $\mathcal{S}(v)$ . If this is the case,  $v'$  is called a *temporary* or *local* variable, and the services in  $\mathcal{S}(v')$  are more likely to remain in the block of rows, making the life of  $v'$  shorter than the size of the block.

In any case, assuming that SCA finds a global minimum, the result of the process of encapsulation depends on, and is determined only by, the given system.

2) *Similarities, cohesion, and coupling*: In an ontological sense, two entities are similar if they share attributes, and the more attributes they share the more similar they are. An entity is *cohesive* if its parts are similar, and two entities are *coupled* if their respective parts share similarities or *uncoupled* if they do not. The notions of similarity, cohesion and coupling are precisely defined and measurable quantities [13]. The attributes of services are their variables. The set  $\mathcal{S}(v)$  of variable-associated services by way of variable  $v$ , share variable  $v$ , and are actually a set of similar services. When SCA encapsulates them into a compact block of rows and drives away other intervening services, it is actually encapsulating similar services and making the block cohesive, and it is also driving away dissimilar services and uncoupling the block of rows from other blocks. SCA terminates only when the profile has reached its minimum and constraints prevent any more compaction from happening. At this point, the capsules in the set created by SCA are as highly cohesive and as weakly coupled as possible.

3) *Objects, classes, and the natural ontology*: The popular definition of object is an association of properties and behavior. But an object must also be perceived as a cohesive individual, a representative of a class, and be sufficiently decoupled from other individuals so it can have a character of its own and an identity. These two features, the individuality and the identity, have made the automation of ontology generation a very difficult goal to achieve.

But SCA solves the problem. The encapsulation process generates highly cohesive, weakly coupled capsules. The capsules contain similar services, and the variables initialized by the services. Each (service, variable) tuple is an elementary object, so it can also be said that the capsules contain elementary objects. Services describe action, behavior, and the variables they initialize represent the properties associated with the behavior, so it can be further said that the capsules indeed associate properties and behavior. The capsules are, therefore, objects, and they are well-defined because they represent highly cohesive individuals, and are sufficiently decoupled from other individuals as to have their own character and identity.

The *class* of an object is defined as the set of domains associated with the variables in the object, and the set of services in the object. Examination of the capsules may reveal identical objects, which belong to the same class. If the object is viewed as a set of elementary objects, then the set of domains in the object is an aggregate of the domains in the elementary objects, and the set of services in the object is an aggregate of the services in the elementary objects. Or, with slight changes in the wording, the object *inherits* the attributes and behavior of the elementary objects ([4], p. 252). Therefore, it can finally be said that SCA generates a set of objects, classifies them into classes, and reveals their inheritance relationships. This is a complete ontology. Once again, and assuming that SCA has found a global minimum, the resulting ontology depends only on, and is determined only by, the given system. This ontology is called the *natural ontology* of the system, and is generated automatically by SCA. No human intervention is required. A domain-based model for classification, subtyping

and type inheritance, including single and multiple inheritance, has been previously proposed ([14], Ch. 12 and ff.), but not automated.

4) *Partitioning the canonical matrix, parallelism, and the encapsulation of flows:* A partition of a matrix is a partition of the set of rows and a partition of the set of columns. The partition is *symmetric* when the matrix is square and the row and column partitions are identical. The process of profile minimization and encapsulation discussed in Section II-F.1 creates identical row and column partitions, induces a symmetric partition of the canonical matrix, and brings many off-diagonal elements closer to the diagonal and into the diagonal blocks. The process results in the diagonal blocks being more densely populated, whereas the off-diagonal blocks in the lower triangle become empty or very lightly populated. The diagonal blocks are square, lower-triangular, with a full diagonal, all diagonal elements equal to  $C$ , and possibly  $A$ 's in the lower triangle. The off-diagonal blocks are rectangular. They are empty in the upper triangle, and possibly contain  $A$ 's but no  $C$ 's in the lower triangle. Column blocks map to objects, row blocks to methods in the objects.

Two blocks are said to be *commutative* if, for every service  $S$  contained in one block and every service  $S'$  contained in the other,  $S$  and  $S'$  are commutative. Conversely, if a service  $S$  can be found in one of the blocks, and a service  $S'$  can be found in the other, such that  $S$  is a predecessor of  $S'$ , then the first block is a *predecessor* of the other. Commutative blocks satisfy Bernstein's conditions and can be *block-commuted* or execute concurrently.

The off-diagonal blocks in the lower triangle represent data flows between the objects. As the profile decreases, so also do the global information flux in the matrix and the average width of the information channel. Information flux becomes more and more localized and grows stronger inside the capsules and weaker across intercapsular boundaries. Information flux encapsulation takes place, and the flux of information is reconciled with the flux of control in a global sense.

The partition of the matrix and the definition of the objects can be achieved by examination of the total inter-row flux  $\Phi(i) = \sum_{j=1}^n \Phi(i, j)$  out of row  $i$  and into row  $i + 1$ , defined by (11), as a function of the row index  $i$ .<sup>3</sup> This function should look like a hill range, with many very narrow and pronounced valleys. The hills identify the capsules themselves, whereas the valleys indicate intercapsular boundaries where flux is very small or zero. Once the horizontal intercapsular boundaries have been identified, the vertical ones are traced symmetrically. For example, in Fig. 1(c), the dotted lines are zero-flux intercapsular boundaries. The published example ([3], Fig. 1(b)) shows 5 intercapsular boundaries, and the other published example ([4], Fig. 1) has 2 intercapsular boundaries that separate the 3 clusters of classified points.

The question of how many valleys are obtained, or how small the information flux is at the valleys, depends on the problem and is a subject for future research. Hopefully, and considering that sparse matrices can be very large, one

would expect a large number of valleys, and many of them to have zero or very small flux. This believe is reinforced because observation confirms the ubiquity of ontologies in the environment, as well as by the enormous success of object-orientation in software engineering. If valleys were scarce, or intercapsular boundaries were too difficult to trace, there would be far fewer objects and ontologies.

Assume first that the flux actually does attain the value of 0 at many valleys and the partition is defined by tracing the intercapsular boundaries at these valleys, and let  $N$  be the *order* of the partition, defined as the number of row blocks or column blocks. Then all off-diagonal blocks are empty and the  $N \times N$  block matrix looks as follows:

$$\Gamma' = \begin{vmatrix} \Gamma_{11} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \Gamma_{22} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \Gamma_{NN} \end{vmatrix} \quad (16)$$

where all elements are block submatrices and  $\mathbf{0}$  represents an empty block submatrix. In this case, all diagonal blocks are uncoupled and commutative, and can execute independently. Execution can start simultaneously and run concurrently in all blocks, because each one of them runs in its own information subspace. Parallelism, the encapsulation of information and control flows, naturally arises. It can be very important if the order  $N$  of the partition is large. This form of parallelism is SCA-related, because it appears as a property of SCA-produced partitions. It can also be considered as a consequence of the fact that the directed *control flow graph* associated with the iMMC is disconnected ([2], §3.4). It is visible in Fig. 1(c), where the two blocks are independent, and in a published example ([3], Fig. 1(b)) where all 6 blocks are variable-independent. There is another type of non-SCA-related parallelism that appears as a result of the transformations in Section III, which can be of an even higher degree.

But the variable-independent blocks do not exhaust SCA's partitioning capabilities. The inter-row flux function  $\Phi(i)$  may still contain plenty of pronounced valleys where the flux is not exactly zero but attains noticeably small values. This makes a further partition of many of the original independent diagonal blocks in (16) possible. Consider one of them, say  $\Gamma_{ii}$ . If additional intercapsular boundaries are traced at the new valleys, a sub-partition of submatrix  $\Gamma_{ii}$  into smaller, but still weakly coupled sub-blocks is obtained. If  $\eta_{ii}$  ( $\eta$  for brevity) is the order of the sub-partition, the resulting  $\eta \times \eta$  block submatrix looks as follows:

$$\Gamma_{ii} = \begin{vmatrix} \gamma_{11} & \mathbf{0} & \cdots & \mathbf{0} \\ \gamma_{21} & \gamma_{22} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{\eta 1} & \gamma_{\eta 2} & \cdots & \gamma_{\eta \eta} \end{vmatrix} \quad (17)$$

Since encapsulation brings off-diagonal elements in the canonical matrix closer to the diagonal, there may be many off-diagonal blocks empty or scarcely populated. The blocks farther away from the diagonal are the most likely to be empty. The information channel is narrow, and most of the information flux is contained in the diagonal blocks. The non-empty

<sup>3</sup>Not to be confused with the total flux crossings  $\Phi(j)$  in column  $j$ , which is a function of the column index  $j$

off-diagonal blocks are weakly coupled because information flux across intercapsular boundaries is weak. Information flux has been encapsulated, and the flow of information and the flow of control are reconciled to the point allowed by the constraints. Complete reconciliation, referred to as a *reunion of information and process*, requires some more work. This is discussed in §III. The sub-partition is *congruent* with the original partition. Therefore, the original partition and all the subpartitions can be merged and considered as one.

5) *The Random SCA and an Introduction to MMC Dynamics*: The original version of the SCA algorithm described in Section II-F is deterministic in nature. Given a canonical matrix with a given permutation, the algorithm starts from that permutation, always the same, scans all services in the given order, determines the commutation range for each one, finds one minimum of the profile without any concern for the possible existence of other minima and following some arbitrary but deterministic rule, and effects the corresponding permutation. The procedure is repeated without change until no further reductions of the profile are found. But the profile of a canonical matrix, when considered as a function of the permutation for the set of all legal permutations, is a complicated function. It may have many global minima, and usually does. It may also have local minima, where the algorithm can stall and not be able to find any of the global minima. This situation leaves a number of questions open regarding the stability and reproducibility of the resulting objects. Are the objects always the same? Do they depend on the particular minimum? What is the meaning of a stall? Can stalls be prevented? Can a found minimum be identified as global or local? Do all objects suddenly appear when the global minimum is reached, or do they or some of them exist in some form *near* a global minimum? How do objects look like in a local minimum? All of which calls for a better investigation of the properties of the profile function.

To make such an investigation possible, a random version of SCA or *Random SCA* was developed. This algorithm starts from a randomly selected, but legal permutation, selects services in a random order, calculates the commutation range for each selected service, finds *all* profile minima in that range, and selects one of them at random. The path followed by the algorithm is different each time, and is not reproducible. The important question is whether the *results* are also different, and if so, what are the differences and what do they mean. Numerical experiments and statistical methods can be used to find answers, giving rise to a new field of research called MMC Dynamics.

Many such experiments were carried out, based on various examples, and some answers have been found and are summarized below. The dynamic views presented below are very different from the traditional understanding of SCA as a mathematical algorithm that minimizes the profile. They are easier to explain if the term profile is replaced with *energy*, which was already done above in Section II-E.4. A brief introduction to the subject will suffice for the purpose of this paper.

- Repeated random runs of the same problem can stall at local minima, but rarely do so. When a minimum is reached,

it is not possible to determine whether the minimum is global or local. It is, instead, possible to backtrack to a previous state with a slightly larger energy, and let SCA run randomly from that state. The algorithm may find its way to a lower minimum, or keep falling repeatedly into the same one or into other minima with the same value of the energy. In this last case, the minima are likely to be global.

- When several global minima are confirmed with reasonable certainty, and the corresponding matrices are examined, it is found that the *same* objects are present. They appear in *different orders* in the matrix, but they are always the same. The objects are stable and reproducible, even though the processes that find them are not. The objects are, indeed, the stable *attractors* that a dynamic MMC system converges to ([4], §4). This experimental fact is very important because it validates the notion that objects are created by energy minimization, and the requirement that objects be stable and reproducible under random conditions is possibly the best definition of object given so far.

- A *theory of objects* is established by these conclusions. The statement of Proposition 3 is the first postulate of the theory. The second postulate is: *The ontology of any finite system represented by a canonical matrix of services is determined by the stable attractors of the profile function*. The theory allows objects to be mechanically and quantitatively generated, following a process that, I believe, is similar to the process followed by the brain, as indicated by the fact that the mechanical objects do indeed appear to be representations of the same natural objects we form in our minds. By contrast, predicate logic and OO programming languages are limited to describing objects that are assumed to be already formed and described in the analyst's mind. The abstraction of mechanically generated objects as they are found in successive steps of the profile minimization algorithm SCA results in an exponential reduction of the complexity of the representation.

- The objects themselves have *states*. The algorithm delivers the objects in their *ground states*. An object is a set of services that appear in a set of adjacent rows of the canonical matrix. The object has an energy of its own. Intra-object commutations between the services in the object may be possible, and may lead to states with the same energy or with different energies. Two states that have the same energy are said to be *degenerate*. When the object is in its ground state, its energy is minimum. Other discrete *energy levels* may be reached by the intra-object commutations that increase the value of the energy, some of them single, some degenerate. Two objects in different degenerate states are still objects of the same class. If their submatrices do not match, they can be permuted as needed to produce a full match without affecting the energy.

- A set of objects contained in a canonical matrix can be viewed as a hot gas of elementary objects contained in a box. The initial temperature of the gas is high. As energy is removed, the temperature is reduced, and the gas starts to condense into larger objects in the box. Local minima of the energy called *potential wells* begin to form, and some objects may fall into them and get captured. Other objects continue to collide and interact with each other, exchanging energy.

When temperature drops further, a global minimum is reached. The global minimum is equivalent to the absolute zero of temperature, where nothing moves and nothing else happens because there is no energy available to make it happen. The process is one of *simulated annealing*, except that the goal is not limited to just finding a global minimum.

- At the core of the theory of MMC Dynamics is the notion that services can be considered as dynamic entities, capable of sensing the environment, competing for the resource of energy, executing on their own when the arguments they need become available, and commuting among themselves to save energy, much like neurons can. This notion makes SCA unnecessary. As previously proposed [4], MMC Dynamics may have a say in the analysis of all kinds of naturally occurring structures that appear as solutions to complex behavior, ranging from atoms and elementary particles to cells, organs, organisms, and societies.

6) *Examples of SCA*: The first case study on SCA is presented in Section V. It is based on a Java computer program used in European universities for teaching refactoring. Other than that, only small examples are available. The reason is that SCA had been proposed only for canonical submatrices in iMMC models, which is a serious limitation. Any large study would have to be restricted to particular cases, and would therefore be domain-dependent or even case-dependent. Undertaking a large effort under such limitations would not be advisable. Instead, motivated by the fact that objects and ontologies seem to appear everywhere with no system or domain restrictions, and convinced that SCA ought to be general, I found ample justification for a theoretical study of the problem. The results, reported in Section III of this publication, indicate that any iMMC, and therefore any finitely realizable physical system, can be represented by a canonical matrix in its entirety. This conclusion makes SCA a universal algorithm.

The small example shown in Fig. 1 is sufficient to illustrate how SCA works. Fig. 1(a) shows a simple, quite disorganized program or set of equations, and Fig. 1(b) shows the corresponding canonical matrix  $\Gamma$ . The vertical profile of  $\Gamma$  is 9, and the data channel has a maximum width of 4 and an average width of 1.5. SCA refactors  $\Gamma$  by permuting services and variables and minimizing the profile, according to (7). The resulting canonical matrix  $\Gamma'$  is shown in Fig. 1(c).  $\Gamma'$  has a vertical profile of 5, the maximum width of the data channel is 2, and the average width is 0.83. The improvement is considerable, even for this small example. The reader can verify that none of the legal permutations in  $\Gamma'$  would reduce the profile any further. The ontology generated by SCA consists of 2 classes with 1 object each. Matrix  $\Gamma'$  is in block-diagonal form. The dotted lines indicate the partitions. All data flows have been encapsulated into the diagonal blocks, and none is crossing the dotted lines. The off-diagonal blocks are empty, indicating that the two objects are independent and can execute in parallel. In the small object,  $h$  is a temporary variable and  $w$  is the output. In the large object,  $p, q$  and  $g$  are temporary and  $s$  is the output. By eliminating the temporary variables, the refactored code can easily be written as shown

in Fig. 1(d), or translated to an object-oriented language with no developer intervention.

Two examples of SCA have been previously published. The first example ([3], p. 191) is in Physics. Ontologies have always been part of the scientific method and scientific expression. Any scientific theory is the natural ontology of the experimental data that support it, built with classes of objects developed by the scientists. Scientific research is a large scale example of ontological engineering and an extensive repository of experience regarding the generation, use and meaning of ontologies. Theories, particularly theories of Physics, provide an excellent testing platform for SCA because of their inherent simplicity and because scientists have exercised great care in constructing them and have tested them very thoroughly. If sufficient experimental information - and of course any supporting previously existing theories - are represented in matrix  $\Gamma$  of (7), SCA should be able to design matrix  $P$  in such a way that  $\Gamma'$  represents the theory.

The example in Physics consists of 18 equations with 30 variables that describe one time step in the accelerated motion of a mass particle under the action of forces in 3 dimensions. However, one point of the example is not to reveal the "meaning" of the equations and variables. It is intended to underscore the notion that meaning is in the ontology itself, and is not a hindrance because any physicist can easily figure it out. In this case, SCA, knowing nothing about meaning, creates a  $6 \times 6$  block-partitioning of the matrix where the 6 diagonal blocks are populated and all off-diagonal blocks are empty. Examination of the partitioning reveals an ontology consisting of 2 classes with 3 objects each. The 2 classes correspond to position and velocity, and the 3 objects in each class correspond to the 3 dimensions of space. The ontology can be interpreted as the law of Independence of the Components of Motion. A very simple law, indeed, but one that is nevertheless taught to every student of Physics. Here, SCA finds the law without human intervention.

The second example ([4], p. 256) considers the classification of points in a plane by proximity into clusters. There are 167 points, and overlapping meshes are used to simulate the continuity of the plane. The resulting canonical matrix is of order 1433, with an initial profile of 299,565. SCA reduces the profile to 15,642 and separates the points into 3 classes, corresponding to 3 clusters.

7) *Summary of SCA*: SCA encapsulates elementary objects to form larger, more meaningful objects. The new objects have more attributes and more behavior than the elementary objects, and are highly *cohesive* because of the strong internal similarities. Mutual coupling is weak or inexistent, because the objects share few or no external similarities. Encapsulated domains *inherit* the services and attributes of their subdomains, giving rise to *inheritance* relationships. In summary, by encapsulating regularities, SCA refactors the canonical matrix, encapsulates services and variables into highly cohesive, weakly coupled capsules, determines a partition of the canonical matrix into objects, reconciles the flow of data with the flow of control, and reveals the *natural ontology* of the model. Refactoring also improves code understandability by encapsulating behavior (services) and properties (variables) and by reducing coupling.

SCA-based algorithms can be used to search information repositories and automatically reveal their natural ontologies.

### III. THE CANONICAL FORM OF THE MMC

#### A. The paradigm

In nature and in human endeavor, resource allocation is always optimized by reusing resources. However, reuse is a constraint, because a resource being used for one purpose is unavailable for another. Reuse imposes an *order* on the set of resources, which may or may not be compatible with certain processes such as parallelism or the development of intelligence. Order is the opposite of parallelism, and is detrimental to intelligence because it is a global property that *requires* intelligence for its implementation.

To resolve the dilemma, I propose a paradigm where an environment favorable for the theoretical analysis or natural onset of such processes is one where order is at a minimum, resources are plenty and not limited by reuse, and the allocation of resources is a *consequence* of the process, not a prerequisite. The present work is a study on the paradigm and on the effects of order.

Because of the universality of the imperative MMC, expressed by Theorem 1, and the ease with which existing system representations can be converted to iMMC format, the iMMC appears to be the ideal model for exploring the paradigm. The iMMC contains four types of order: the partial order imposed on the services by the predecessor-successor constraints, the partial order of the services imposed by the lives of variables, the total order on the services imposed by the matrix of sequences  $Q$ , and the partial order imposed on the values in each domain by the requirement that a variable can have a single value at a time. They all depend on the particular execution path, that is, on the control variables. The total order is a linear extension of all other orders, because it is compatible with and contains all of them, but the only authentic order is the partial order of the predecessor-successor constraints. The remaining three orders, including the total order, are superfluous. They will disappear when their cause, the reuse of resources, is eliminated as part of the process of application of the paradigm.

Sequential computer programs share many features with the iMMC, among them the three partial orders and the total order that contains them. The human analyst responsible for the total order must guarantee its compatibility with all three partial orders. But the partial orders are invisible in the code, and the only way for the programmer to comply is to constantly trace the execution paths. A nearly impossible task that makes sequential programming difficult and prone to errors, and correct parallelism difficult to attain. Violations of the partial orders have frequently resulted in catastrophe. Scientists who develop theories also confront similar circumstances, and sometimes take decades to reach their conclusions, but the analysis and testing of scientific theories are much more rigorous than those of software.

The first step in the application of the paradigm to the iMMC is to identify and eliminate all forms of reuse while preserving the universality and overall behavior of the model,

where “overall” means the same final results, not necessarily the same intermediate results or the same procedures. This is a lengthy, but very rewarding process. When it is completed, what emerges is the *canonical form* of the MMC, or cMMC, where the entire system is represented by a canonical matrix. Because of the parallelism and the ontological and self-organizing capabilities of canonical matrices discussed above, it would appear that such a result is highly desirable and encouraging, and should be interpreted as an indication that the track is right. Furthermore, as the process evolves, several brain-like features also appear. They include robustness, the autonomy of services, which leads to parallelism and awareness, and the ability to learn and to increment the learning ability by reuse of acquired knowledge, which reveals intelligent behavior and is therefore key for the intelligence to grow, and the reunion of data and behavior. Awareness and parallel pattern recognition have also been attributed to quantum phenomena [15]. Since the MMC is a general model of computation, not a model of the brain, this result is unexpected but still encouraging and again suggestive of a right track. In view of all of which I propose:

**Proposition 3.** *Every finitely realizable physical system can be perfectly represented by a Matrix Model of Computation in canonical form.*

This proposition is examined in depth below, and an extensive case study is presented. I believe the proposition is a theorem, but since it has not been proved yet it will remain as a working hypothesis. The transformations discussed below profoundly alter the structure of the model while preserving its overall behavior, properties, and universality, and lead to the iMMC  $\rightarrow$  cMMC conversion. Table I summarizes the transformations, and Fig. 2 shows a simple example. The transformations are general and apply to all systems, but in practical cases simplifications are frequent and some of the transformations may not be needed.

#### B. Transformation MC. The elimination of mutators and multiple codomains

Services with mutators and/or more than one codomain perform more than one function at a time and are a form of service reuse. Therefore, transformations must be provided to eliminate all mutators and to convert all multiple-codomain services into single-codomain services in matrix  $C$ . Let  $S : \widehat{\mathcal{D}}_A \times \widehat{\mathcal{D}}_M \rightarrow \widehat{\mathcal{D}}_M \times \widehat{\mathcal{D}}_C$  be a service where  $\widehat{\mathcal{D}}_M$  is the set of all mutators in  $S$ . If  $m = |\widehat{\mathcal{D}}_M| > 0$ , then service  $S$  has  $m$  mutators that must be eliminated. To eliminate the  $m$  mutators, for each  $D \in \widehat{\mathcal{D}}_M$  define a new domain  $D' = \{d \mid d \in D\}$  and the mapping:

$$D \rightarrow D' : d \in D \text{ maps to } d' \in D' \iff d = d' \quad (18)$$

Equation (18) describes a set  $\mathfrak{R}$  of  $m$  new services, each of them with one argument and one codomain, and no mutators. To complete the transformation, replace service  $S$  with the  $m$  new services in set  $\mathfrak{R}$ , followed by the following modification of service  $S$ :

$$S' : \widehat{\mathcal{D}}_A \times \widehat{\mathcal{D}}' \rightarrow \widehat{\mathcal{D}}_M \times \widehat{\mathcal{D}}_C \quad (19)$$

TABLE I  
SUMMARY OF TRANSFORMATIONS

Name	Section	Purpose	Method
MC	III-B	Eliminate mutators and multiple-codomain services.	Service conversion to single-codomain.
SV	III-C	Eliminate service re-invocation and variable re-initialization.	Execution search.
AS	III-D	Eliminate the matrix of sequences and control variables.	Grant autonomy to services.

where all domains on the left-hand side of the mapping are arguments, and all those on the right-hand side are codomains because the domains in  $\mathcal{D}_M$  are no longer mutators. Finally, change the matrix of sequences  $\mathbf{Q}$  as needed to link the new services in the indicated order. For example,  $x := x+3$ , where  $x$  is a mutator and  $m = 1$ , would be replaced with  $z := x$ ,  $x := z + 3$ , where  $x$  is no longer a mutator. Once all mutators have been eliminated from matrix  $\mathbf{C}$ , the definition of service, Equation (4), is simplified as follows:

$$S : \widehat{\mathcal{D}}_A \rightarrow \widehat{\mathcal{D}}_C \quad (20)$$

The fragment of C code shown Fig. 2(a) is an example of path-aware routing. Because the example is so simple, it can be shown directly in code and matrix  $\mathbf{Q}$  can be omitted. There is only one control variable,  $i$ , and one execution path. Variables 0, 1, 2, 3 are input data, and are assumed to be already initialized. Variables  $i$  and  $a$  are mutators, and there are no multiple codomains. Fig. 2(b) shows the code after the mutators have been eliminated by introducing the new variables  $j$  and  $b$ .

To convert multiple-codomain services into single-codomain services, let  $S : \widehat{\mathcal{D}}_A \rightarrow \widehat{\mathcal{D}}_C$  be a service where  $\widehat{\mathcal{D}}_C$  is the set of all codomains in  $S$  and the mutator-free definition of service (20) has been used. If  $k = |\widehat{\mathcal{D}}_C| > 1$ , then service  $S$  has  $k > 1$  codomains and must be replaced with a set of  $k$  single-codomain services. The elements of set  $\widehat{\mathcal{D}}_C$  are  $k$ -tuples, with their elements taken from the domains in  $\mathcal{D}_C$ , and the effect of service  $S$  is to select one of the  $k$ -tuples by assigning a truth value to it. However, since the  $k$ -tuples are all different, and all possible combinations of elements are present, assigning a truth value to one of the  $k$ -tuples is the same as assigning a truth value to each one of the elements of the  $k$ -tuple in their respective domains. Thus, service  $S$  can be replaced with a set of  $k$  single-codomain services defined as follows:

$$\mathfrak{R} : \{S' \mid S' \text{ is the service } \widehat{\mathcal{D}}_A \rightarrow D', D' \in \mathcal{D}_C\} \quad (21)$$

where  $|\mathfrak{R}| = k$ . A service  $\widehat{\mathcal{D}}_A \rightarrow D'$  in  $\mathfrak{R}$  has  $D'$  as its only codomain. A preliminary definition for a service  $S'$  in (21) can be obtained by assuming  $S'$  to be identical to  $S$ , except that, on output, all codomain initializations are thrown away except for the codomain  $D'$  of service  $S'$ . However, since  $D'$  may contain less information than  $\widehat{\mathcal{D}}_C$ , it may happen that not all arguments in  $\mathcal{D}_A$ , or not all calculations described in the submodels of service  $S$ , are actually needed to effect the mapping, resulting

in dead arguments in the service declaration and dead code in the service definition. These unnecessary arguments and code must be identified and removed on a case by case basis. Once all mutators and multiple-codomain services have been eliminated from matrix  $\mathbf{C}$ , the definition of service (20) is simplified as follows:

$$S : \widehat{\mathcal{D}}_A \rightarrow D_C \quad (22)$$

where  $D_C \in \mathcal{D}_C$  is the single codomain in the service. Equation (22) can also be written in convenient functional notation:

$$S : y = f(x_1, x_2, \dots), \quad (23)$$

where  $y$  is the codomain and  $x_1, x_2, \dots$  are the given arguments. An example for this transformation would be a service  $S$  that describes the calculation of a column vector  $\mathbf{a} = \mathbf{A}\mathbf{b}$  as the product of matrix  $\mathbf{A}$  and another column vector  $\mathbf{b}$ . Service  $S$  sets all elements of  $\mathbf{a}$ . If the size of  $\mathbf{a}$  is  $n$ , then service  $S$  has  $n$  codomains. If  $n > 1$ , then  $S$  must be replaced with  $n$  single-codomain services, each of which calculates one single element of  $\mathbf{a}$ . Of course, each single-codomain service requires as arguments only one row of elements of matrix  $\mathbf{A}$ , and the elements in the remaining rows are unnecessary and should not be arguments for that service.

### C. Transformation SV. The elimination of service re-invocation and variable re-initialization

In the course of iMMC execution, any service can be reached multiple times by convergent joints in matrix  $\mathbf{Q}$  or by invocation from another service that has been reached multiple times, and any variable can be re-initialized multiple times by a service that has been invoked more than once and/or by different services. Multiple invocation is a form of service reuse, and variable re-initialization is a form of variable reuse. A re-initialized variable has many lives and establishes a partial order for the services that reference the lives.

Both forms of reuse must be eliminated. The elimination is achieved by performing an execution search (§II-C) and replacing services, variables and sequences encountered along the execution path with uniquely identified copies. Let  $\mathcal{M} = (\mathbf{C}, \mathbf{Q})$  be the given iMMC model in the form obtained after transformation MC has been completed. The execution search can be performed directly in  $\mathcal{M}$ , or a control flow graph can be constructed and used for the search. In either case, the graph or the model are assumed to be connected, or if they are not, each connected part is treated separately. Let  $E =$

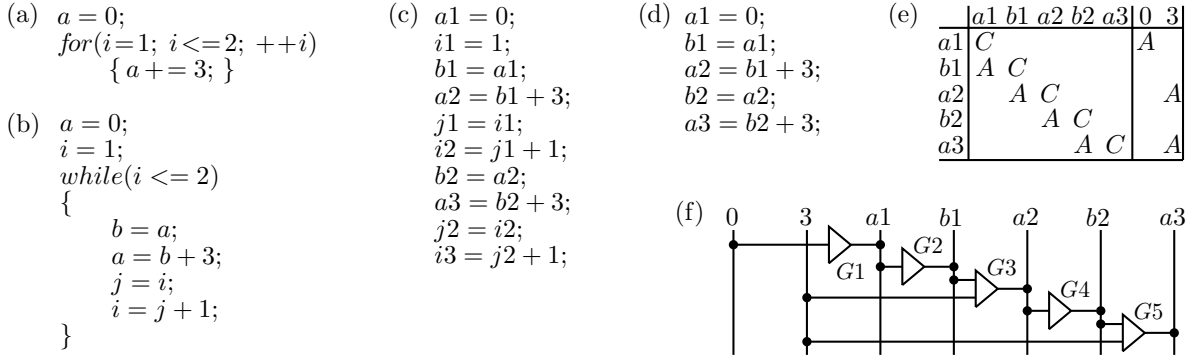


Fig. 2. A running example for Sections III-B to III-E. The single control variable  $i$  provides path-aware routing. (a) A fragment of C code that calculates  $2 \times 3$ . (b) Transformation MC. Variables  $b$  and  $j$  have been introduced to remove mutators. (c) Transformation SV. The execution path, with variables renamed and a total of 11 services. (d) The same execution path with the control variables and unused code removed. (e) Transformation AS. The canonical matrix with autonomous services. (f) The equivalent abstract machine, with 5 AND gates.

$\{e \mid e \text{ is an execution path in } \mathcal{M}\}$  be the set of all execution paths in  $\mathcal{M}$ . The purpose of the procedure is to construct a new equivalent model  $\mathcal{M}' = (C', Q')$ . Matrix  $C'$  is constructed with multiple copies of the services and variables in  $C$ , each with its own unique identifier, and matrix  $Q'$  is constructed with multiple copies of the tuples in  $Q$ , also with their own unique identifiers. All paths in  $E$  must be searched. If  $n_Q \times m_Q$  and  $n_C \times m_C$  are the sizes of  $Q$  and  $C$ , and  $n_{Q'} \times m_{Q'}$  and  $n_{C'} \times m_{C'}$  are the sizes of  $Q'$  and  $C'$ , respectively, then, in general,  $n_{C'} \leq n_C |E|$  and  $n_{Q'} \leq n_Q |E|$ , because not all services in  $C$  are present in every execution path.

Initially,  $C'$  is empty, and  $Q'$  is initialized with the usual columns  $\mathcal{P}$  and  $\mathcal{F}$  (§II-B), and with an additional column  $\mathcal{E}$ , the domain of  $E$ , but otherwise empty. Matrix  $C'$  is discussed first. For each  $e \in E$ , let  $S$  be a service in  $C$  encountered by the search along path  $e$ . Since  $S$  has a single codomain and no mutators, it can be expressed in functional notation, as in (23):

$$S : c = f(a_1, a_2, \dots, a_j). \quad (24)$$

At the time  $S$  is reached, and irrespective of the state of initialization of codomain variable  $c$ , a copy of  $c$  is made, say  $\gamma(c)$ , and set in correspondence with a new column of  $C'$ . Variable  $\gamma(c)$  is labelled with the triple  $(e, n, c)$ , where  $n$  is the sequential number assigned by the search, and designated to be the *current* copy of  $c$ . The domain for variable  $\gamma(c)$  is  $\{true, false\}$ , and, since it has not been initialized yet, its current value is *false*. This variable represents the state of initialization of  $c$ . At the time  $S$  is reached, all its argument variables are in the initialized state, and for each one of them one or more copies have been made at the time of initialization. Let  $\bar{a}_i$  be the current value of argument  $a_i, i = 1, \dots, j$ . Then

$$\bar{c} = f(\bar{a}_1, \dots, \bar{a}_j), \quad (25)$$

where  $\bar{c}$  is the value to be assigned to  $c$ . Now, let  $\gamma(a_i)$  be the current copy of variable  $a_i$ . Copy  $\gamma(a_i)$  was made precisely when variable  $a_i$  was initialized with value  $\bar{a}_i$ , and, as a consequence, the current value of  $\gamma(a_i)$  is *true*.

To continue with the search, service  $S$  is executed, and variable  $c$  is initialized with the value  $\bar{c}$ . To represent the equivalent action in model  $\mathcal{M}'$ , a copy of  $S$ , say  $\gamma(S)$ , is

made and labelled with the triple  $(e, n, S)$ , where  $n$  is the sequential number assigned by the search, and merged into a new row of matrix  $C'$ . The copy is defined by:

$$\gamma(S) : \gamma(c) = f'(\gamma(a_1), \dots, \gamma(a_j)), \quad (26)$$

where  $f'$  represents the mapping from the arguments to the codomain. However, the current values of the arguments are all *true*, and the only effect of  $\gamma(S)$  is to set the codomain  $\gamma(c)$  to *true* as well. Therefore, service  $\gamma(S)$  can be symbolically represented as the following logical conjunction:

$$\gamma(S) : \gamma(c) = \text{AND}(\gamma(a_1), \gamma(a_2), \dots, \gamma(a_j)). \quad (27)$$

Equation (27) defines the generic function  $f$  used in (23). The names of the values, such as  $\bar{a}_i$  or  $\bar{c}$ , are irrelevant in  $\mathcal{M}'$ . The values themselves have been coded into the structure of the model, and are described only by their functionality. Besides that, the only thing that matters is whether they are present or not.

Equation (27) is the pivotal equation for this paper. The net effect of transformation SV is that, for each service encountered on each path  $n \geq 1$  times,  $n$  copies are made and merged into  $C'$ . And, for each variable initialized  $m \geq 1$  times,  $m$  copies are made and inserted into  $C'$  as part of the processing of the services. The result is that all services are logical conjunctions, all different and invoked only once, all variables are different and initialized only once, matrix  $C'$  contains exactly one  $C$  in each row and each column, and the domain representation is in the expanded form (§II-A.2).

Matrix  $Q'$  is now discussed. Each time the execution search reaches a service  $P$ , the usual copy  $\gamma(P)$  is made, and the search then proceeds to the sequence of  $P$  in matrix  $Q$ . The sequence consists of 1 or more tuples, and a subset, say  $V$ , of 0 or more control variables, all of which are in the initialized states. The search now finds the unique tuple in the sequence that matches the current values of all the control variables in  $V$ , and the corresponding service  $F$ , the next destination of the search. If  $F$  has been visited before, there exist copies of it, and one of them, say  $\gamma(F)$ , is current. Otherwise, a copy is made and becomes  $\gamma(F)$ .

Let  $k = |V|$ , let  $v_1, \dots, v_k$  be the control variables in  $V$ , and let  $\bar{v}_1, \dots, \bar{v}_k$  be the corresponding current values. The matching tuple, say  $t$ , is given by:

$$t : (P, \bar{v}_1, \dots, \bar{v}_k, F). \quad (28)$$

Tuple  $t$  translates into tuple  $\gamma(t)$ , given by:

$$\gamma(t) : (e, \gamma(P), true, \dots, true, \gamma(F)), \quad (29)$$

where *true* appears  $k$  times, in correspondence with the  $k$  variables. Tuple  $\gamma(t)$  alone constitutes the entire sequence of service  $\gamma(P)$ . It is labelled with a triple containing the execution path identifier, the identifier of the original tuple  $t$ , and the number assigned to tuple  $\gamma(t)$  by the search, and merged into matrix  $Q'$ . The search continues as service  $F$  becomes the next  $P$  and gets executed. In all,  $n \geq 1$  copies of  $P$  are made, the sequence of  $P$  begets exactly  $n$  sequences, one per copy, each with exactly one  $(k + 3)$ -tuple, and each with a different set of control variables. They are all merged into  $Q'$ , and a one-one bijective correspondence is established between tuples in  $Q'$  and services in  $C'$ .

Finally, to every  $e \in E$  there corresponds exactly one sub-collection of input data sets (§II-C). They are all transformed accordingly, to correspond to the new variables, and each subcollection is renamed with a tuple that includes the original identifier and the identifier of path  $e$ .

1) *Discussion of transformation SV:* After transformations MC and SV are completed, the resulting matrix  $C'$  contains no  $M$ 's, and exactly one  $C$  in each row and each column. All forms of branching, joints, cycles, and reused code have disappeared. Matrix  $Q'$  now has the additional control variable  $e$ , which induces a partition of  $Q'$  into submatrices, each corresponding to one execution path and each completely independent of the others. This independence leads to massive parallelism. If resources were available, it would be possible to execute all paths at once. If not, then studies can be conducted to determine how best to share the available resources, or what minimum resources are necessary to avoid degrading the parallelism too much. The parallelism, in turn, implies that all possible solutions become available simultaneously, which produces the equivalent to *awareness*, because it involves the maximum possible amount of knowledge about a problem. Full parallelism, however, has not been attained yet, because a total order still applies to the services along each execution path. Full parallelism will only be achieved when autonomy is granted to services by transformation AS below (§III-D).

There have been some concerns about transformation SV, particularly in regard with the possibly very large number of execution paths. The number of input data sets may be very large, even nominally infinite, but there are reasons why the number of paths may still be quite manageable and the size of the paths themselves may be small in many cases.

- Paths are determined by control variables, not input variables. If  $D_1, \dots, D_n$  are the control domains, then the number of paths is  $|D_1| \dots |D_n|$ . The cardinalities of the control domains are frequently small. Besides, there are many ways to reduce the number of control variables themselves. For example, a loop with a control variable can be linearized

before attempting the search. Even a parameterized loop can be linearized provided the appropriate notation is introduced. Just like we humans do it.

- No new services, variables or tuples are generated in the course of search, only copies are made. The copies do not have to be physically made. Secluded paths do not have to be replicated at all.

- Execution traces tend to be short. Only portions of the total code are used, and the logic is excluded. The logic overhead is usually very heavy in a traditional program.

- A subsequent transformation, transformation AS, grants autonomy to services and eliminates matrix  $Q$  and all its control variables and the code needed to calculate them. The resulting code is actually *shorter* than the original one.

- Depending on the application, it may or may not be necessary to generate the entire set of paths.

- The iMMC (or the CFG) can be subdivided, for example into strong components, or into secluded paths and joints, or into sections shared by more than one path. The subdivisions can be used to create a level structure, which can be subsequently used to partition and better manage the set of paths.

- A very large number of execution paths may be an indication of poor system design, perhaps too many parts with little in common. But a system can not be reliable unless all its parts are reliable, and analysis is necessary anyway. Way too often this problem is left for developers to address.

There is also a concern regarding *feedback*. There appears to be a confusion with the use of the word *same*. Feedback occurs when the output signal of some device is combined with an incoming signal to create the actual input to that device, and is considered essential for intelligence to develop. Feedback causes the *same* variable – the input signal – to be repeatedly reused by re-initializing it to different values, whereas transformation SV appears to preclude feedback by eliminating reuse and re-initializations. However, a feedback loop always includes a time delay unit, and the input signal can indeed have different values but *not at the same time*. Transformation SV simply implies that the input signal will be represented not with one variable which is a function of time, but with a sequence of variables that correspond to the values of the input signal at different instants of time.

In the example of Fig. 2, the execution path is shown in (c), with the variables renamed. The triple  $(e, v, n)$  is simplified to “ $vn$ ”, because there is only one execution path. Each statement represents a service. The matrix of sequences is not explicitly given.

#### D. Transformation AS. The elimination of the matrix of sequences and the autonomy of services

A no-mutators single-codomain service is said to be *autonomous* (§II-D.2) when its execution is determined solely by the state of initialization of its arguments. An autonomous service remains inactive while any of its arguments are uninitialized, and executes on its own and initializes its codomain as soon as all the arguments become initialized. The arguments act as *signals* that *trigger* the execution of the service.

Transformation AS applies to the model  $\mathcal{M}' = (C', Q')$ , obtained after transformation SV has been completed. Transformation AS consists of eliminating matrix  $Q'$  and granting autonomy to all services in matrix  $C'$ . Let  $\mathcal{M}'' = (C'')$  be the resulting system. System  $\mathcal{M}''$  contains many autonomous services. The self-execution of a service in  $\mathcal{M}''$  causes a previously un-initialized variable to become initialized, and may in turn trigger the self-execution of other services that were waiting for that variable, resulting in a *cascade* of self-executions. In  $\mathcal{M}''$ , cascades are constrained only by the predecessor-successor conditions, which are built into the services themselves. Because of the bijective correspondence between services and variables in  $C''$ , cascades are completely independent of each other and *race conditions* can not occur. Cascades represent the highest degree of parallelism and awareness attainable in a given system.

The elimination of the matrix of sequences makes all the control variables unnecessary. They must also be eliminated from matrix  $C''$ , along with the services that initialize them and any associated code. However, only pure control variables must be eliminated. A variable that appears in a service as an argument can not be eliminated even if it also appears as a control variable in the matrix of sequences. The elimination of the control variables amounts to a complete elimination of all explicit logic. The logic becomes hardcoded in the services themselves, and results in great simplification. More importantly, the logic becomes *local*, meaning under the local control of the services themselves, and implies a full reconciliation of the flows of data and control. These properties are considered essential for the onset of intelligent behavior.

Transformation AS completes the conversion iMMC  $\rightarrow$  cMMC from a fully designer-controlled system to a fully autonomous AI system that can design itself, program itself, and evolve on its own. Transformation AS completes the process of analysis, and the process of synthesis begins. Synthesis is concerned with modularization and the reuse of resources for specific purposes, such as learning and artificial intelligence, parallelization, the construction of ontologies by the SCA algorithm, and others. Attempting to synthesize a system without the analysis would be like trying to build it out of generic parts. Not very advisable.

The order of services in matrix  $C''$  is irrelevant because it does not affect the order in which they execute. However, in preparation for the process of synthesis, there is one more optional step that should be considered: permuting the services in a *legal* order. A legal order is any total order compatible with the partial order of the predecessor-successor constraints. A legal order is required for SCA to operate, and it can be easily obtained by application of the sorting algorithm of section IV-B.

Fig. 2(d) shows the example code after the control variable and associated code have been eliminated, and the matrix of sequences has been eliminated as well by making the services autonomous. The corresponding canonical matrix is shown in (e).

### E. The abstract machine

Equation (27) and transformation AS define a service in the cMMC as a logical conjunction driven by a set of 1 or more digital input signals, and capable of producing a digital output signal whenever all the input signals are present.

The cMMC is a virtual machine, one that requires hardware or wetware to run on. An *abstract machine*, instead, is a device that can, in principle, be built, although it is not intended to be. It can be built because it does not contradict the laws of Physics, and since it is a machine it is fit to have physical properties. A simple abstract machine can be defined based on the cMMC. The machine has a set of buses, usually represented with vertical lines, that carry the signals. It also has a set of AND gates, in correspondence with the autonomous services, each with one single output connected to the bus that carries the codomain signal for that service, and one or more inputs connected to the corresponding argument signals. The machine is similar to a circuit-based intelligent agent.

This machine is equivalent to the cMMC and functions in the same way. Initially, an external actor activates some buses, the input data. This causes a cascade of activations similar to the one in the cMMC, until a combination of output buses get activated. This machine can be built, and it can learn, and grow, and develop intelligence.

Fig. 2(f) shows the abstract machine for the example code shown in (a). Assuming 0 and 3 are initially given, gate  $G1$  activates variable  $a1$ . Gates  $G3$  and  $G5$  sense the signal at 3, but remain inactive. The activation of  $a1$  causes  $G2$  to activate  $b1$ , and this time  $G3$  activates  $a2$ . Finally,  $b2$  and  $a3$  get activated in succession.

## IV. BASICS OF MMC SUPERVISED LEARNING

The transformations of Section III provide a direct link between the canonical MMC and existing forms of highly organized system descriptions, such as computer software, business or UML models, technical manuals, administrative forms, or scientific theories and literature. Training the MMC from organized knowledge is, in itself, a form of learning. In the case of software, for example, the link would be the translation of computer programs to iMMC format, which can be done by a suitably modified parser, followed by a conversion from iMMC to cMMC by application of the transformations of Section III.

But the subject of this Section is a more conventional form of training, *supervised learning*, or learning with a teacher, where the *teacher* has knowledge about the subject being taught and is capable of providing instructions and communicating with the MMC, but may impart arbitrary information on arbitrary subjects. An extensive case study presented in Section V includes a detailed example of this type of learning.

### A. Teacher instructions and the learning algorithm

Traditional error correction learning schemes, such as pattern-matching methods, rely on the ability of a prescribed model to “automatically” interpolate among a set of master patterns learned and stored by the model. However, the design of the model is usually guided by considerations other than the

requirements of interpolation. For example, the basic design of a Hopfield network for the recognition of handwriting depends on the network being amenable to Lyapunov modeling and on the availability of suitable stable attractors, and no so much on the details of handwriting recognition. Under such circumstances, it becomes difficult to adjust the design to the actual needs of the application. In the MMC, instead, the process of information acquisition and the processes of information organization, interpolation, and control, are not linked and do not overlap, but are completely independent. Each one can be separately understood and optimized. This is an essential advantage.

Canonical MMC learning is very special, because of two unique features. First, predecessor-successor relationships among the services establish a partial order among them. If the order is not satisfied, the matrix is not canonical, but a simple sorter can reestablish the order and canonize the matrix. Second, SCA can organize the knowledge into existing classes of objects or define new ones as needed. Because of these unique features, a machine based on the model should in principle be able to acquire and accumulate knowledge on any subject and learn any task without limitations in the degree of detail, even if the teacher supplies the information in an arbitrary order, much like a child learning at school. MMC supervised learning is in the form of a dialog between the teacher and the machine, where the teacher supplies instructions and the machine asks questions to which the teacher may or may not reply. Unsupervised learning from percepts is also possible but not discussed here. The knowledge base is stored in a *Learning Matrix*  $L$ , which is an MMC matrix of services that allows the services to be in any arbitrary order *without* a matrix of sequences. As usual, the rows of  $L$  describe services, and the columns describe variables. Each row contains exactly one  $C$  and at least one  $A$ , and each column contains 0 or one  $C$  and 0 or more  $A$ 's, but there must be at least one  $A$  or one  $C$  in each column. The act of initializing a variable by a service is equivalent to assigning a value of *true* to the conjunction of the variables in that service, as is done for sentences in Propositional Logic. There are no mutators in  $L$ . Initially, matrix  $L$  is empty. At any point during the execution of the learning algorithm, matrix  $L$  has the following form:

$$L = \begin{vmatrix} G & H \\ J & K \end{vmatrix} \quad (30)$$

where submatrix  $G$  is canonical, submatrices  $H$  and  $J$  contain only  $A$ 's, and submatrix  $K$  contains  $A$ 's and  $C$ 's.

An *instruction* is a statement issued by the teacher that is interpreted as the definition of a service, i.e., a mapping, in accordance with (27). When an instruction is acquired, it is converted into a service and inserted at the bottom of the learning matrix, causing submatrices  $J$  and  $K$  to expand. This elementary process is called an *act of learning*. The notation for teacher instructions resembles simple Prolog clauses. In the instruction, the codomain name is listed on the left, an equals sign follows, and then the names of the arguments are listed in any order, separated by spaces and without repetition. For example:

$$a = b \ c \ d \quad (31)$$

defines the service

$$\gamma(S) : a = \text{AND}(b, c, d) \quad (32)$$

with codomain  $a$  and arguments  $b, c$ , and  $d$ , all of them over the domain  $\{true, false\}$ . Instructions are given in any order and do not have to be organized by subject. The basic learning algorithm operates as follows:

- (P3.1) Initially, the learning matrix is empty. In general, at any point during the process, the learning matrix is structured as in (30).
- (P3.2) Check the input. If an instruction is available, continue. If not, go to step 4. Optionally, if the matrix needs updating, go to step 5.
- (P3.3) Parse the instruction, define a service, and store the service at the bottom of the learning matrix.
- (P3.4) Find the set of un-initialized variables, select any one of them, and request an explanation for that variable. If no un-initialized variables are found, output "OK" and go to step 2. Optionally, if the matrix needs updating, go to step 5.
- (P3.5) Apply the sorting algorithm (§IV-B) to sort the services into a partial order compatible with the predecessor-successor constraints. The entire learning matrix must be sorted, since newly acquired information may affect the canonical submatrix  $G$ . The sorting algorithm converts the learning matrix again to the form of (30).
- (P3.6) If submatrix  $K$  is not empty, apply SCA and the matching algorithms to form classes of objects. The output of this step is the natural ontology corresponding to the information acquired so far. Adjust  $G$ ,  $H$  and  $J$  as necessary.
- (P3.7) Output the new class definitions to the teacher, and go to step 2 to continue the learning process, but now in terms of the new objects.

Note that while any un-initialized variables remain, the machine will keep asking questions, a behavior that closely resembles our own insatiable curiosity.

Learning by *modular aggregation*, or *aggregate learning*, is a mode of learning where entire modules of knowledge are simply aggregated to form a larger module, by means of multiple acts of learning, followed by SCA optimization and possibly additional training to teach the machine how to use the new modules. Aggregate learning is possible because of the independence of services in the cMMC. For example, if a machine has learned how to play chess and another how to play checkers, the two cMMC's can be aggregated to form a new machine that can play both games, and still learn another.

### B. The sorting algorithm

As teacher instructions keep coming in an arbitrary order, and the corresponding services are assembled into matrix  $L$  without regard for the predecessor-successor constraints, an illegal matrix may be obtained where variables are used before being initialized. An example can be seen in (39), as part of the case study. The purpose of the sorting algorithm is to restore the partial order required to satisfy the constraints.

This is always possible if there are no circular constraints, because each service initializes exactly one variable and the set of predecessors is uniquely defined by the arguments. The basic sorting algorithm is designed to be applied to the entire learning matrix, although more advanced versions that take advantage of stable objects will appear in the future. The output from the sorting algorithm is  $L$  expressed in the form of (30), in terms of the four submatrices, where submatrix  $G$  is canonical and ready for SCA refactoring. Let the size of  $L$  be  $n \times m, m \geq n$ . As usual, matrix  $L$  has exactly one  $C$  in every row, and 0 or 1  $C$ 's in every column, and all nonzeros are either  $A$  or  $C$ . At the beginning of step  $k, k = 1, \dots, n$ , the submatrix of order  $k - 1$  located in rows  $(1, k - 1)$  and columns  $(1, k - 1)$  of  $L$  is canonical. The purpose of step  $k$  is to form a canonical submatrix of order  $k$  in rows  $(1, k)$  and columns  $(1, k)$  of  $L$ . Step  $k$  of the algorithm is as follows:

- (P4.1) Determine set  $\Gamma = \{c \mid \text{the variable in column } c \text{ is currently initialized}\}$ . Thus,  $\Gamma$  is the set of all columns of  $L$  that contain a  $C$ .
- (P4.2) Let  $S_r$  be the service in some row  $r, r = k, \dots, n$  of  $L$ , and let  $\Gamma_r = \{c \mid c \geq k, L_{rc} = A\}$  be the set of all columns outside  $G$  where service  $S_r$  has  $A$ 's.
- (P4.3) Find a row, say row  $\bar{r}$ , such that  $\Gamma \cap \Gamma_{\bar{r}} = \emptyset$ . Thus, service  $S_{\bar{r}}$  has no  $A$ 's in any initialized column at or beyond column  $k$ . If such a row does not exist, the algorithm terminates. Otherwise, let  $\bar{c}$  be the column that contains the codomain of service  $S_{\bar{r}}$ , and continue.
- (P4.4) Permute service  $S_{\bar{r}}$  from row  $\bar{r}$  to row  $k$  and column  $\bar{c}$  to column  $k$  of  $L$ , and redefine submatrices  $G, H, J$ , and  $K$  accordingly. This concludes step  $k$ .

## V. CASE STUDY

A case study is an analysis in depth of a particular case, designed to help the reader experience the theory at work. It should be neither too complex nor too simplistic, but it must contain just enough detail to demonstrate how the various techniques fit and act together. Computer simulations of systems are a good source of examples for MMC case studies. I selected the Java version of a Local Area Network (LAN) simulation ([16], [17]) used in many European universities to teach refactoring. It is GNU software and it is just the right size to fit in journal pages. Section V-A addresses the issues considered in this case study, the system description is presented in Section V-B, and the analysis begins in Section V-C.

### A. Issues considered in this case study

Because of the very wide range of potential applications of MMC theory, the range of issues that can be addressed in a case study is also very wide. Of particular interest are issues that concern AI itself, and, since the example selected is one of software, those that concern software engineering and the application of AI to software engineering. The case study attempts to shed some light on the following questions: Can the MMC self-organize? Can SCA create an ontology from a given set of services, without using any additional semantic

information? Can the ontology be applied to develop an object-oriented (OO) model of the system, and eventually to generate a computer program in an OO programming language? Are SCA-designed objects stable in a dynamic environment? Are they reproducible? How do SCA-designed classes compare with human-designed classes? How can the canonical model be trained, how much, and in what detail? Is it scalable? Can the canonical model be programmed? Can it serve as if it were a programming language? Can MMC become the foundation of a new MMC-centric software development environment capable of automating tasks such as refactoring code and translating between programming languages?

This study focuses on the self-organization of a system described by simple instructions provided by a teacher in no particular order. The machine converts the instructions into services and stores them into a learning matrix. Occasionally, a sorter sorts the services into a partial order compatible with the predecessor-successor constraints and partitions the matrix into four submatrices, one of which is canonical, and the SCA algorithm determines the natural ontology of the canonical submatrix. At that point the teacher can learn the new classes created by the machine and use them to compose more advanced instructions and continue teaching. The system self-organizes by creating the classes of objects, and advances its ability to learn because of its ability to self-organize. The system, in fact, gains intelligence.

Since the selected example is a Java program with a given ontology and precise sequencing, it is first necessary to remove all object-oriented structures and any sense of order. The first is accomplished in Section V-C by execution search (§II-C), where all functions that involve objects are replaced with the detailed code that supports those functions. The search results in a set of 9 execution paths written in single-assignment C. The elimination of order is accomplished by selecting the teacher's instructions from the execution paths at random. The approach resembles that of a school teacher who already knows the model and the objects and expects his pupils to synthesize the same objects from his teachings. One important advantage of the approach is that it makes it possible to compare the resulting machine-generated ontology with the original one, prepared by a human analyst.

Objects and supervised learning are interrelated subjects, because each one depends on the other. I have chosen to cover objects first, in Section V-D and V-E, and supervised learning last, in detail, in Section V-F.

### B. System description

The network is a token ring with several workstations, three in this case, interconnected by serial transmission lines. Each station has two message buffers, one for input and one for output, connected by a parallel bus. The input buffer is also connected to the incoming line, and the output buffer to the outgoing line. A station can create a message and copy it to its output buffer, from where it is transmitted to the input buffer of the next station in the ring. Upon receiving the message, and depending on its source and destination, a station can take appropriate action such as print and destroy it or send it to the next station.

The Java program is given in Fig. 3(a). Class *Message* represents a message, with source, destination, and contents. Class *Station* is a generic station, whose basic behavior is to *accept()* a message in its input buffer and *send()* it to its output buffer for transmission to the next station. Classes *NetworkTester*, *PrintServer*, and *Workstation* extend *Station* and specialize its behavior by overriding *accept()*, and contain methods that allow them to construct *Message* objects and take other actions based on the source and destination of a message. The token ring consists of a *Workstation* (0), a *PrintServer* (1), and a *NetworkTester* (2), and is described by the bijective mapping  $\{0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0\}$  of the set of stations onto itself, where each station maps to the next station. This class, say *TokenRing*, supports functionality such as *GetNext()* and *GetPrinter()*. Yet another class, say *TransmissionLine*, supports the transmission of messages between stations. The last two classes are irrelevant for the purpose and are not included.

### C. Preliminary analysis

As outlined in Section III, the analysis of a general system consists of two major steps: obtaining a representation of the system in terms of a canonical matrix, and applying SCA techniques to it in order to generate the natural ontology of the system. To accomplish the first step, an imperative MMC model of the system is first developed and the transformations of table I are applied to it.

The analysis is greatly simplified when code is available, as in this case, because the execution search can be performed directly in code by a modified parser, and there is no need to generate the graph, or even the iMMC model. Furthermore, examination of the code in Fig. 3(a) reveals that there are no mutators and no multiple-codomains, so transformation MC is not necessary, and one can proceed directly to transformation SV, the execution search. The only control variables are the source and the destination of a message. Assuming that each station can send messages to every other station, including itself, there are 3 possible values for each control variable, resulting in a total of 9 execution paths. All execution paths must be obtained and analyzed. In this case, the execution paths were obtained manually by examination of the program. As an example, the path where *NetworkTester* sends a message to itself and prints its contents when it receives it back, is given in Fig. 3(b). This code is already in single-assignment form and contains no re-initializations of variables. The notation *parallel*{...} is a reminder that the bus connecting the input and output buffers of each station is parallel, and therefore the total order of the execution path does not apply to the statements between braces.

The elimination of object references may result in very long names, which do not fit well in figures representing large sparse matrices. For the purpose of presentation, the long names have been replaced with much shorter names. Examples of shorthands are *dst* for *m.destination* and *od0* for *W.outBuffer.destination*. The shorter names make the code somewhat more difficult to understand, but this is in fact an advantage because it helps the reader to concentrate on the

mechanics of SCA without being distracted by the “meaning” of names. For the same reason, I have chosen not to explain the names. My working hypothesis is that the functionality contained in the program alone is sufficient to determine the objects, and that any meaning associated with the objects is a *consequence* of the functionality, not a prerequisite. OO techniques actually solve the *inverse* problem: given the meaning, find the objects that satisfy that meaning. SCA, instead, solves the *direct* problem: given the functionality, find the objects and their meaning. And SCA does not use names.

*Existence conditions* are paramount in formal mathematics, but in systems analysis they are sometimes considered obvious and simply omitted. For example, a network must exist before a message can be sent over it, and the message must exist before it can be sent. Existence conditions induce a partial order on the services that participate in the construction and the application of an object, *in addition* to the predecessor-successor constraints. In OO code, an existence condition appears implicitly every time an object is constructed. If the OO code is translated into sequential non-OO code, the existence conditions are preserved in the total order of the statements. However, if the order is destroyed, as it will be in this case, the partial order must be explicitly enforced by inserting a *placeholder* variable that represents the existence condition. The placeholder is later removed when an object that satisfies the existence condition is defined. Existence condition placeholders are present nearly everywhere, even in documents, but seldom noticed. For example, the previous sentence implies that a document must exist before a placeholder can be placed in it.

The code obtained from Fig. 3(b) when object references are eliminated and all long names are replaced with abbreviated names, is given in Fig. 4(a), formatted as teacher’s instructions (§IV-A). The domain for all variables is  $\{true, false\}$ . The initial value of each variable is *false*, or un-initialized, and the variable gets initialized when its value is changed to *true*. Each statement defines a service where the variable on the left is the codomain and the variables on the right are the arguments. When the arguments become all initialized, the service executes on its own and initializes its codomain. However, for the purpose of argument, assume that the total order still applies. The 6 initial statements represent users entering initialization information for the stations. For example, *st0* is the identifier and *tp0* is the type for station 0. Since the user of station 0 can not enter both values at the same time, and in preparation for the elimination of the total order, service *st0* has been made a predecessor of service *tp0*. The same applies to the other 2 stations. However, it is perfectly possible for 3 different users to be typing at the 3 stations simultaneously, so this possibility is left open. Services *msg* and *ntw* enforce the existence conditions (§V-C) for the message and the network, respectively, and services *rd0*, *rd1* and *rd2* guarantee that the message transmitted over the serial lines has arrived in its entirety before it is accepted by the station. Variables *W*, *w*, *P*, *p*, *T*, *t*, and *NTN* are input constants.

The code in Fig. 4(a) represents the teacher’s knowledge about the system. The teacher selects the instructions at random, and imparts them to the machine one at a time. At the

```

(a) public class Message{
    public String source;
    public String destination;
    public String contents;
}
public class Station{
    public String stationID;
    public String type;
    public Message inBuffer, outbuffer;
    public void accept(){
        outBuffer = inBuffer;
        send();
    }
    public void send(){
        GetNext().inBuffer = this.outBuffer;
        GetNext.accept();
    }
    public void print(String s){
        System.out.println(s);
    }
    protected Station GetNext(){...}
    protected PrintServer GetPrinter(){...}
}
public class NetworkTester extends Station{
    public void testNetwork(){
        Message m = new Message();
        m.source = stationID;
        m.destination = stationID;
        m.contents = "NTNT";
        outBuffer = m;
        send();
    }
    public void accept(){
        if(inBuffer.source == this)
            print(inBuffer.contents);
        else super.accept();
    }
}
public class PrintServer extends Station{
    public void accept(){
        if(inBuffer.destination == this)
            print(inBuffer.contents);
        else super.accept();
    }
}
public class WorkStation extends Station{
    public void originate(){
        Message m = new Message();
        m.source = stationID;
        m.destination = GetPrinter().stationID;
        m.contents = "NTNT";
        outBuffer = m;
        send();
    }
    public void accept(){
        if(inBuffer.source == this)print("ERR");
        else super.accept();
    }
}

(b) //Construct and initialize the stations.
WorkStation W;
PrintServer P;
NetworkTester T;
W.stationID = "W";
W.type = "w";
P.stationID = "P";
P.type = "p";
T.stationID = "T";
T.type = "t";

//Station T constructs a message to itself.
Message m;
m.source = T.stationID;
m.destination = T.stationID;
m.contents = "NTNT";

//Copy to T.outBuffer.
parallel{
    T.outBuffer.source = m.source;
    T.outBuffer.destination = m.destination;
    T.outBuffer.contents = m.contents;
}

//T.GetNext() returns W. Copy to W.inBuffer.
W.inBuffer.source = T.outBuffer.source;
W.inBuffer.destination = T.outBuffer.destination;
W.inBuffer.contents = T.outBuffer.contents;

//Copy to W.outBuffer.
parallel{
    W.outBuffer.source = W.inBuffer.source;
    W.outBuffer.destination = W.inBuffer.destination;
    W.outBuffer.contents = W.inBuffer.contents;
}

W.GetNext() returns P. Copy toP.inBuffer.
P.inBuffer.source = W.outBuffer.source;
P.inBuffer.destination = W.outBuffer.destination;
P.inBuffer.contents = W.outBuffer.contents;

//Copy to P.outBuffer.
parallel{
    P.outBuffer.source = P.inBuffer.source;
    P.outBuffer.destination = P.inBuffer.destination;
    P.outBuffer.contents = P.inBuffer.contents;
}

//P.GetNext() returns T. Copy to T.inBuffer.
T.inBuffer.source = P.outBuffer.source;
T.inBuffer.destination = P.outBuffer.destination;
T.inBuffer.contents = P.outBuffer.contents;

//Network OK.
T.print(T.inBuffer.contents);

```

Fig. 3. (a) The original Java program. (b) The execution trace for the case where NetworkTester sends a message to itself.

end of the process, which is described in great detail in Section V-F, and after application of the sorting algorithm (§IV-B) that restores a partial order compatible with the predecessor-successor constraints, the machine has generated the  $33 \times 33$

matrix of Fig. 5. The matrix is canonical, but has a large profile and shows no identifiable patterns. This matrix is the actual input for the SCA process. The corresponding code, given in Fig. 4(b), is legal, but disorganized and unrecognizable. If a

```

(a) st0 = W
    tp0 = w st0
    st1 = P
    tp1 = p st1
    st2 = T
    tp2 = t st2
    src = st2
    dst = st2
    cnt = NTN
    //message complete
    msg = src dst cnt
    //network complete
    ntw = st0 st1 st2 tp0 tp1 tp2
    os2 = src msg ntw
    od2 = dst msg ntw
    oc2 = cnt msg ntw
    is0 = os2
    id0 = od2
    ic0 = oc2
    //message received
    rd0 = is0 id0 ic0
    os0 = is0 rd0
    od0 = id0 rd0
    oc0 = ic0 rd0
    is1 = os0
    id1 = od0
    ic1 = oc0
    //message received
    rd1 = is1 id1 ic1
    os1 = is1 rd1
    od1 = id1 rd1
    oc1 = ic1 rd1
    is2 = os1
    id2 = od1
    ic2 = oc1
    //message received
    rd2 = is2 id2 ic2
    prt = ic2 rd2

(b) st2 = T
    cnt = NTN
    st0 = W
    st1 = P
    dst = st2
    src = st2
    tp2 = t st2
    tp1 = p st1
    tp0 = w st0
    msg = src dst cnt
    ntw = st0 st1 st2 tp0 tp1 tp2
    os2 = src msg ntw
    is0 = os2
    od2 = dst msg ntw
    oc2 = cnt msg ntw
    id0 = od2
    ic0 = oc2
    rd0 = is0 id0 ic0
    os0 = is0 rd0
    od0 = id0 rd0
    is1 = os0
    id1 = od0
    oc0 = ic0 rd0
    ic1 = oc0
    rd1 = is1 id1 ic1
    os1 = is1 rd1
    oc1 = ic1 rd1
    is2 = os1
    od1 = id1 rd1
    ic2 = oc1
    id2 = od1
    rd2 = is2 id2 ic2
    prt = ic2 rd2

(c) st1 = P
    tp1 = p st1
    st0 = W
    tp0 = w st0
    st2 = T
    tp2 = t st2
    ntw = st0 st1 st2 tp0 tp1 tp2
    dst = st2
    src = st2
    cnt = NTN
    msg = src dst cnt
    oc2 = cnt msg ntw
    od2 = dst msg ntw
    os2 = src msg ntw
    id0 = od2
    is0 = os2
    ic0 = oc2
    rd0 = is0 id0 ic0
    os0 = is0 rd0
    oc0 = ic0 rd0
    od0 = id0 rd0
    is1 = os0
    ic1 = oc0
    id1 = od0
    rd1 = is1 id1 ic1
    os1 = is1 rd1
    od1 = id1 rd1
    oc1 = ic1 rd1
    id2 = od1
    is2 = os1
    ic2 = oc1
    rd2 = is2 id2 ic2
    prt = ic2 rd2

```

Fig. 4. Various forms of the execution trace of Fig. 3(b), formatted as teacher instructions. The functional dependencies are preserved, but the total order and all object-oriented structures of the original code are eliminated. The trace shown in (a) still preserves the total order but has no objects. The teacher's instructions were selected from this trace at random, thus destroying the original order. Trace (b) corresponds to the canonical submatrix generated by the learning process and shown in Fig. 5. The code in trace (b) is legal, but scrambled and unrecognizable. Trace (c) is the final refactored code, corresponding to the minimum-profile canonical submatrix of Fig. 6 obtained by the SCA algorithm.

developer were asked to refactor this code and create an OO design, he would surely know how to do it. Developers think along lines very similar to SCA, and they refactor by detecting similarities and constricting scopes, but they do not write the matrix explicitly, can not calculate or minimize profiles, and can not tell local minima from global minima. The SCA process is discussed next.

#### D. Profile minimization and matching

The  $33 \times 33$  canonical matrix obtained as the output of the supervised learning process, shown in Fig. 5, has a profile of 272. The  $A$ -counts by row and by column are shown next to the matrix. There appear to be no visible patterns, either in the  $A$ -count arrays nor in the matrix itself. It would be very difficult to find any useful matches.

The random SCA algorithm was used to minimize the profile of the matrix from Fig. 5. Numerous runs were performed in order to assess convergence (§II-F.5). All runs but one converged to different legal permutations of the same objects, with a minimum profile of 209. The remaining run stalled

at a local minimum with a profile of 211, but a repeat run converged again to 209. The numerical experiments provide strong evidence that 209 is indeed the global minimum. It is not the only global minimum, because the objects can and do appear in many different permutations, but they are always the same objects and convergence is satisfied. One of the resulting minimum-profile matrices is shown in Fig. 6. Many patterns are clearly visible in this matrix. Matching must now be performed in order to identify the classes and objects.

The only valid match between two submatrices is a full match between all their elements, which is a two-dimensional match. There are a number of simpler 1D matches, which are not full matches but serve as useful indicators for finding the more difficult 2D matches, including the arrays formed by counting the number of  $A$ 's in each row or column, the scopes of the variables, either vertical or horizontal, and the intensities of data flows from row to row or column to column. The  $A$ -counts depend on the presence of  $A$ 's outside of the diagonal blocks, so they may not match even if the submatrices do, but they can still provide useful clues for finding the submatrices.

	s	c	s	s	d	s	t	t	t	m	n	o	i	o	o	i	i	r	o	o	i	i	o	i	r	o	o	i	o	i	r	p		N				
	t	n	t	t	s	r	p	p	p	s	t	s	s	d	c	d	c	d	s	d	s	d	c	c	d	s	c	s	d	c	d	d	r	T				
	2	t	0	1	t	c	2	1	0	g	w	2	0	2	2	0	0	0	0	0	1	1	0	1	1	1	1	2	1	2	2	2	t	N				
st2	C																																	A	1			
cnt		C																																		A	1	
st0			C																																	A	1	
st1				C																																A	1	
dst	A				C																																1	
src	A					C																															1	
tp2	A						C																													A	2	
tp1								A																													A	2
tp0									A																												A	2
msg										A																												3
ntw	A										A																											6
os2												A																										3
is0													A																									1
od2														A																								3
oc2															A																							3
id0																A																						1
ic0																	A																					1
rd0																		A																				3
os0																			A																			2
od0																				A																		2
is1																					A																	1
id1																						A																1
oc0																							A															2
ic1																								A														1
rd1																									A													3
os1																										A												2
oc1																											A											2
is2																												A										1
od1																												A										2
ic2																													A									1
id2																														A								1
rd2																															A							3
prt																															A							2
	4	2	2	2	2	2	2	1	1	1	3	3	1	2	1	1	2	2	3	1	1	2	2	1	2	2	1	2	3	1	1	1	1	2	1	1	0	

Fig. 5. The 33 × 33 canonical matrix obtained by random supervised teaching from the code in Fig. 4(a), and corresponding to the scrambled code of Fig. 4(b). The profile of this matrix is 272. The matrix of input data is attached on the right-hand side. The A-counts by column and row are shown next to the matrix. This matrix does not show any identifiable patterns.

Another good way for finding partitions and matchings is by examination of the inter-row or inter-column flux functions defined in (11), as discussed in Section II-F.4.

All objects in Fig. 6 are in their ground states, but profile-preserving intra-object permutations are still possible, as discussed in Section II-F.5. Therefore, matching attempts between candidate submatrices must not be restricted to the form in which they appear, but must also consider all other combinations with the same profile.

The goal of profile minimization is to consolidate the objects so that the matching algorithms can find them. The goal of partitioning by submatrix matching is to obtain the largest possible matching submatrices. Frequently, the matching submatrices can be further partitioned based on coupling. This time the goal is to obtain the smallest matching submatrices that best encapsulate coupling. The overall goal is to block-diagonalize the matrix as best as possible, leaving as few A's outside the blocks as possible. The blocks represent objects, and the A's outside them are couplings. Matching can begin by looking for prominent features in one of the indicators, and trying to expand them as much as possible. In this case, the bottom A-counts contain the prominent

feature (3) repeated several times, which can be immediately expanded to the left to form (2, 2, 2, 3), but no further, and then to the right to form (2, 2, 2, 3, 1, 1, 1). This immediately suggests the partition marked with solid lines as a good candidate. None of the three submatrices match as they are, but a simple analysis demonstrates that the permutation (1,2,3,4,5,6,7,8,9,10,11,13,14,12,15,16,17,18,21,19,20,24,22, 23,25,27,26,28,29,30,31,32,33), which still has a profile of 209, results in a full match. A different approach would be to notice that the 4 × 4 and 3 × 3 principal diagonal blocks of the three submatrices, do fully match, leaving some A's outside the blocks as couplings among them. This leads to the partition marked with dashed lines. The same dotted partition can be obtained if the initial feature (3) is expanded to the left only, yielding two occurrences of (1, 1, 1, 2, 2, 2, 3), and the corresponding submatrices (12, 18) and (19, 25), which expand to (26, 32). It might seem that the solid-line partition and the dashed-line partition are different, but when they are subpartitioned for coupling, the same result is obtained. Separately, but following similar procedures, the partition (1, 2), (3, 4), (5, 6) is also obtained, as is shown in the figure with dashed lines.



Fig. 7(a), and the corresponding MMC object model appears in Fig. 7(b). As the figure indicates, the attributes of class *S* are *st* and *tp*, whereas those of class *M* are *dst*, *src*, and *cnt*. The existence conditions discussed in Section V-C, represented by services *ntw*, *msg*, *rd0*, *rd1* and *rd2*, are not used in the object model.

These results correspond to the execution trace  $NT \rightarrow NT$ . A similar analysis of the remaining 8 traces yields similar class models with the same objects, confirming the validity of the classes. The rest of the analysis relies on well known object-oriented methods.

Objects *S0* and *S1* in Fig. 7(b) are initialized from input data but never used. Object *S2* is also initialized from input data. An examination of the detail in Fig. 6 reveals that the data in *S2.st2* is used to initialize *M0*, and then travels unchanged through objects *B2*, *L0*, *B0*, *L1*, and *B1*, all the way to object *L2*. Another piece of input data is used to initialize *M0.cnt* and travels along the same path, but this time all the way to *prt*. A ladder structure such as this represents a *traveling object* of class *M*. It defines a total order for the set of objects along the path  $\{S2, M0, B2, L0, B0, L1, B1, L2, prt\}$  and creates a directed association between every pair of consecutive objects, such as  $(B2, L0)$ ,  $(L0, B0)$ ,  $(B0, L1)$ ,  $(L1, B1)$ , and  $(B1, L2)$ . Other paths confirm these associations and define additional ones. For example, a path from the Workstation to itself finds the association  $(L2, B2)$ , which closes a loop and creates a circular order on the set of objects  $\{L0, B0, L1, B1, L2, B2\}$ , all of them of class *Message*. The corresponding class could be called *TokenRing*. It supports the circular mapping through a method that could be called *GetNext()*. Other paths associate  $(L0, prt)$ ,  $(L1, prt)$ , which, when combined with  $(L2, prt)$ , indicate that *TokenRing* should also support a *GetPrinter()* method. Furthermore, the last 3 associations indicate that *L0*, *L1*, *L2* are similar among themselves but different from *B0*, *B1*, *B2*, thus inducing a partition of the set into subsets  $\{L0, L1, L2\}$  and  $\{B0, B1, B2\}$ . The subsets give rise to classes *InputBuffer* and *OutputBuffer*, both of them extensions of class *Message*.

The fact that *M0* in Fig. 6 is a traveling object composed from input data and data in *S2*, and immediately used to initialize *B2*, suggests that *S2* should support this functionality through a suitable method, which might be called *testNetwork()*, in addition to methods *send()* and *accept()* in class *Station*. This makes *S2* an instance of an extension of class *Station*, say class *NetworkTester*. A similar analysis with other paths leads to other extensions of class *Station*, say classes *PrintServer* and *Workstation*. A complete translation of the MMC object model into Java would be heavily language-dependent and will not be attempted. But a comparison of the results above with those in Section V-B indicates that the SCA process has generated an ontology of classes and objects with virtually identical object-oriented information as the original description of the system. I propose that this information is sufficient to support such a translation. A UML object model can be created, following previously discussed guidelines ([3], §4), and a conversion to any OO programming language can operate from there. An MMC-centric development environment has been proposed ([2], §4),

which would include modules for translation between MMC and various languages, and therefore between the languages themselves.

The object model in Fig. 7 is a canonical matrix, where the values of all variables are either *true* or *false* and the services act as AND switches. This model has the same mathematical properties as the original model of Fig. 5, and the same procedures can be applied to it. For example, the teacher can continue to teach, but now in terms of objects in the current object model, and SCA can be applied and a new higher-level ontology of classes can be obtained, the elements of which are the current objects. The system has, in fact, self-organized and gained some intelligence, and is ready to use it and gain some more. In a more general context, a canonical system that constantly receives information from a teacher or from sensory perception, and where SCA constantly runs in the background and updates the ontology, has been proposed ([2], §3).

F. Supervised Learning in the Case Study

The basics of MMC supervised learning are discussed in Section IV. This Section presents a dialog between a teacher and an MMC machine, where the teacher imparts an instruction and the machine constructs a service, stores it into the learning matrix, and asks for the definition of one of the un-initialized variables. Since there are always un-initialized variables, the machine keeps asking questions and never replies “OK”, a behavior that resembles that of a small child. The teacher can answer the question or ignore it and continue with other subjects. At a certain point, the machine refactors its current learning matrix and forms an ontology of objects. From that point on the teacher can use the objects to communicate with the machine and further train it. The dialog is presented as an indefinite sequence of steps, and each step includes one instruction received from the teacher, the resulting learning matrix, and the question asked by the machine. At some point the teacher pauses, allowing the machine time to refactor the learning matrix, after which the teacher continues imparting instructions. The instructions come from the execution trace of Fig. 4(a), but the teacher imparts them in an arbitrary order. The objective is to train the machine to “understand” the classes and objects defined in the canonical matrix of Fig. 6. Initially, the learning matrix is empty.

**Teacher:**  $tp2 = t\ st2$

**Learning matrix:**

$$\begin{array}{c|cc} & t & s \\ & p & t \\ tp2 & C & A & A \end{array} \tag{33}$$

**Machine:**  $st2 ?$

**Teacher:**  $st2 = T$

**Learning matrix:**

$$\begin{array}{c|ccc} & t & s & \\ & p & t & T \\ tp2 & C & A & A \\ st2 & & & C & A \end{array} \tag{34}$$

**Machine:**  $t ?$

**Teacher:**  $tp1 = p st1$

**Learning matrix:**

	$\frac{t}{p}$	$\frac{s}{t}$	$\frac{t}{p}$	$\frac{s}{t}$
$tp2$	$C$	$A$	$A$	
$st2$		$C$	$A$	
$tp1$			$C$	$A$

**Machine:**  $st1 ?$

**Teacher:**  $st1 = P$

**Learning matrix:**

	$\frac{t}{p}$	$\frac{s}{t}$	$\frac{t}{p}$	$\frac{s}{t}$	$P$
$tp2$	$C$	$A$	$A$		
$st2$		$C$	$A$		
$tp1$			$C$	$A$	$A$
$st1$				$C$	$A$

**Machine:**  $p ?$

**Teacher:**  $os0 = is0 rd0$

**Learning matrix:**

	$\frac{t}{p}$	$\frac{s}{t}$	$\frac{t}{p}$	$\frac{s}{t}$	$\frac{o}{s}$	$\frac{i}{s}$	$\frac{r}{d}$	$\frac{o}{s}$
$tp2$	$C$	$A$	$A$					
$st2$		$C$	$A$					
$tp1$			$C$	$A$	$A$			
$st1$				$C$	$A$			
$os0$					$C$	$A$	$A$	

**Machine:**  $is0 ?$

**Teacher:**  $is0 = os2$

**Learning matrix:**

	$\frac{t}{p}$	$\frac{s}{t}$	$\frac{t}{p}$	$\frac{s}{t}$	$\frac{o}{s}$	$\frac{i}{s}$	$\frac{r}{d}$	$\frac{o}{s}$	$\frac{i}{d}$	$\frac{i}{c}$
$tp2$	$C$	$A$	$A$							
$st2$		$C$	$A$							
$tp1$			$C$	$A$	$A$					
$st1$				$C$	$A$					
$os0$					$C$	$A$	$A$			
$is0$						$C$				$A$

**Machine:**  $rd0 ?$

**Teacher:**  $rd0 = is0 id0 ic0$

**Learning matrix:**

	$\frac{t}{p}$	$\frac{s}{t}$	$\frac{t}{p}$	$\frac{s}{t}$	$\frac{o}{s}$	$\frac{i}{s}$	$\frac{r}{d}$	$\frac{o}{s}$	$\frac{i}{d}$	$\frac{i}{c}$
$tp2$	$C$	$A$	$A$							
$st2$		$C$	$A$							
$tp1$			$C$	$A$	$A$					
$st1$				$C$	$A$					
$os0$					$C$	$A$	$A$			
$is0$						$C$		$A$		
$rd0$							$A$	$C$	$A$	$A$

**Machine:**  $os2 ?$

During this learning process the teacher chose to answer the machine's questions 4 times and ignored them twice to impart

instructions of his choice. The resulting learning matrix of (39) is not a legal matrix of services. Assume now that SCA and the sorting algorithm (§IV-B) run in the background, and that after (39) the teacher pauses, allowing enough time for the machine to sort and refactor the training matrix, or perhaps to go to sleep and let the algorithms to do their work in the background. The sorting algorithm takes 7 steps, but details are left for the reader. One possible result is:

(35)

	$\frac{s}{t}$	$\frac{s}{t}$	$\frac{t}{p}$	$\frac{t}{p}$	$\frac{i}{s}$	$\frac{r}{d}$	$\frac{o}{s}$	$\frac{o}{s}$	$\frac{i}{d}$	$\frac{i}{c}$
$st2$	$C$							$A$		
$st1$		$C$							$A$	
$tp2$	$A$		$C$				$A$			
$tp1$		$A$		$C$					$A$	
$is0$					$C$					$A$
$rd0$						$A$	$C$			$A$
$os0$							$A$	$A$	$C$	

(40)

which is to be compared with (30). This sorted learning matrix is now legal and contains a  $7 \times 7$  canonical submatrix in the first 7 rows and 7 columns, which can be SCA-refactored. When this is done, the result is:

(37)

	$\frac{s}{t}$	$\frac{t}{p}$	$\frac{s}{t}$	$\frac{t}{p}$	$\frac{i}{s}$	$\frac{r}{d}$	$\frac{o}{s}$	$\frac{o}{s}$	$\frac{i}{d}$	$\frac{i}{c}$
$st2$	$C$							$A$		
$tp2$	$A$	$C$					$A$			
$st1$			$C$						$A$	
$tp1$				$A$	$C$					$A$
$is0$						$C$				$A$
$rd0$							$A$	$C$		$A$
$os0$								$A$	$A$	$C$

(41)

The  $7 \times 7$  canonical matrix is now refactored and profile-minimized, and it already shows 2 classes with a total of 3 objects. Objects ( $st2, tp2$ ) and ( $st1, tp1$ ) have been discussed before. Their submatrices match with each other, and belong to the same class. The third object is ( $is0, rd0, os0$ ), which at this time is an entirely valid object and defines its own class. We know, however, from the discussion in Section V-E and Fig. 6, that this object will not survive. This is normal, because forming objects is an ongoing process. As information keeps arriving, new objects are formed, and existing objects are modified or replaced with new ones. Scientists know this phenomenon, they are always prepared to change their theories in the light of new observations if any incompatibilities are found. Software maintenance engineers know that too.

At this point, the machine can inform the teacher about its objects, and from this point on the teacher can start using the objects to better communicate with the machine and to continue the training. The machine has, in fact, gained intelligence.

When the training continues beyond the last training matrix, (39), by either entering what remains of the 33 statements of the code of Fig. 4(a) in any arbitrary order, or by using the new objects to simplify communication, the result is a  $33 \times 33$  non-canonical training matrix. This training matrix corresponds to illegal code, but when it is canonized by the sorting algorithm,

the  $33 \times 33$  sparse matrix of Fig. 5 is obtained, which is now canonical but highly disorganized, and corresponds to the legal but scrambled code shown in Fig. 4(b). Finally, after SCA refactorization, the canonical minimum-profile matrix of Fig. 6 is obtained, which corresponds to the code of Fig. 4(c), and where matching can be performed to form more objects and arrive at the MMC object model of Fig. 7(b), as discussed above. And the process repeats itself, since now the teacher can use the updated objects that the machine understands and continue the training indefinitely.

### G. Implementation

Some colleagues have asked questions regarding how to implement the sparse matrices and the various algorithms described in this paper. MMC is universal, and as such, all required data structures and algorithms can and will one day be implemented as part of the MMC model itself, making it self-organizing and self-refactoring. For now, however, I developed a basic implementation in C++ which is very briefly described here.

A sparse matrix is implemented as two lists, one of *Service* objects and one of *Variable* objects. The elements of the matrix are described by *Role* objects. Each *Role* object knows its *Service* and its *Variable*, and each *Service* and each *Variable* knows the list of their *Roles*. This cross-indexing is invariant under permutations, which greatly simplifies matters. Base class *SparseMatrix* and several derived classes *CanonicalMatrix*, *InputMatrix*, *TrainingMatrix*, etc, describe the matrices themselves, and deal with input, output, verification and related tasks with the help of a *Reader* class. At the time where SCA is to be applied and more efficiency is needed, the lists of *Services* and *Variables* are converted to arrays. A very light *Permutator* class, which knows the arrays but not the matrices, takes care of the assignment of *Services* to rows and *Variables* to columns, as well as of all tasks related to permutations and profile calculations.

## VI. CONCLUSIONS AND OUTLOOK

The MMC is a general theory of systems. If the brain is a system, and I believe it is, then the brain, including the human brain, should work like the MMC, possibly a combination of its various forms, iMMC, cMMC, and aMMC. If so, then everything we can be intelligent about, including all our theories and thinking, and even MMC theory itself, should be amenable to MMC representation. In other words, the MMC would be a general representation that can handle all of human knowledge.

If this is true, then the MMC should be very general, and the study of its properties would be very important. It should be possible to apply it with success to virtually every area of human endeavor one can think about. Such a question can only be answered through experimentation and observation, and through the work of many scientists. On my part, I can only submit the very limited findings discussed in this paper.

However, the MMC was not intended to be a model of the brain, and should not be judged only in terms of its ability to simulate brain function. In particular, it would be a mistake to

compare MMC's current capabilities with humans'. There is a long road ahead before MMC can do anything comparable with what humans can.

The next obvious step is to start training MMC models. The model I propose for MMC development is similar to nature's model for the development of the brain. In nature, DNA is used to make copies of brains, but infants must later learn how to recognize images and measure distances, and children must learn how to multiply. In the case of the MMC, making copies is easy, and aggregate learning can be used to merge them with other MMC models, followed as usual with SCA processing to integrate the system. It should be possible to start several parallel efforts and combine the results into a higher level MMC, of which copies will later be made and combined into an even higher level MMC. Inter-MMC communications should also be developed, so that MMC's can share their knowledge. Human-MMC communications should also be developed, both ways, so that humans and MMCs can share their knowledge. None of this is easy, but it may be within reach.

## VII. APPENDIX. BACKGROUND ON DIRECTED GRAPHS

In this Appendix, a brief background on directed graphs, strong components, connectivity level structures and depth-first search is presented. This material is fundamental for the discussion in Section III-C. A more extensive background and references can be found in [6], Chapter 5. The example shown in Fig. 8(a) helps to explain the definitions that follow.

A *directed graph* or *digraph*  $G = (V, E)$  is a set of *vertices*  $V$  and a set of *edges*  $E$ . An edge  $e \in E$  is an ordered pair of distinct vertices  $e = (u, v)$ , where  $u, v \in V$ . The direction of the edge is from vertex  $u$  to vertex  $v$ . Edge  $(u, v)$  is said to *leave* vertex  $u$  and *enter* vertex  $v$ , or to *lead* from  $u$  to  $v$ . It is also said that edge  $(u, v)$  is an *incoming* edge for vertex  $v$  and an *outgoing* edge for vertex  $u$ . The *indegree* of a vertex is the number of edges that enter it, and the *outdegree* is the number of edges that leave it. A *path* in a directed graph is an ordered set of vertices  $u_1, u_2, \dots, u_{n+1}$  such that  $(u_i, u_{i+1})$  is an edge in  $E$  for  $i = 1, 2, \dots, n$ , and  $n$  is the *length* of the path. The path can also be regarded as an ordered set of edges  $(u_1, u_2), \dots, (u_n, u_{n+1})$ . Vertices  $u_1$  and  $u_{n+1}$  are the *ends* of the path, and the remaining vertices are *internal* vertices. A path is *secluded* if every internal vertex has only one incoming edge and one outgoing edge, both of which belong to the path, and it can not be enlarged without losing that property. The path is *simple* if it does not intersect or overlap itself. A simple path contains no repeated vertices or edges. A secluded path is always simple.

Two given vertices  $u, v$  are *connected* by a path if a path exists in  $G$  with  $u$  and  $v$  as its end points. If such is the case, then  $v$  is said to be *reachable* from  $u$ . A *directed cycle* or simply *cycle* is a path with at least two vertices which begins and ends at the same vertex. A directed *subgraph*  $G' = (V', E')$  of  $G = (V, E)$  is a graph that consists of some or all vertices of  $G$  and some edges of  $G$ , or  $V' \subseteq V, E' \subseteq E$ . A *section graph* of  $G$  is a subgraph that contains exactly all edges  $(u, v)$  such that both  $u$  and  $v$  belong to the subgraph. A

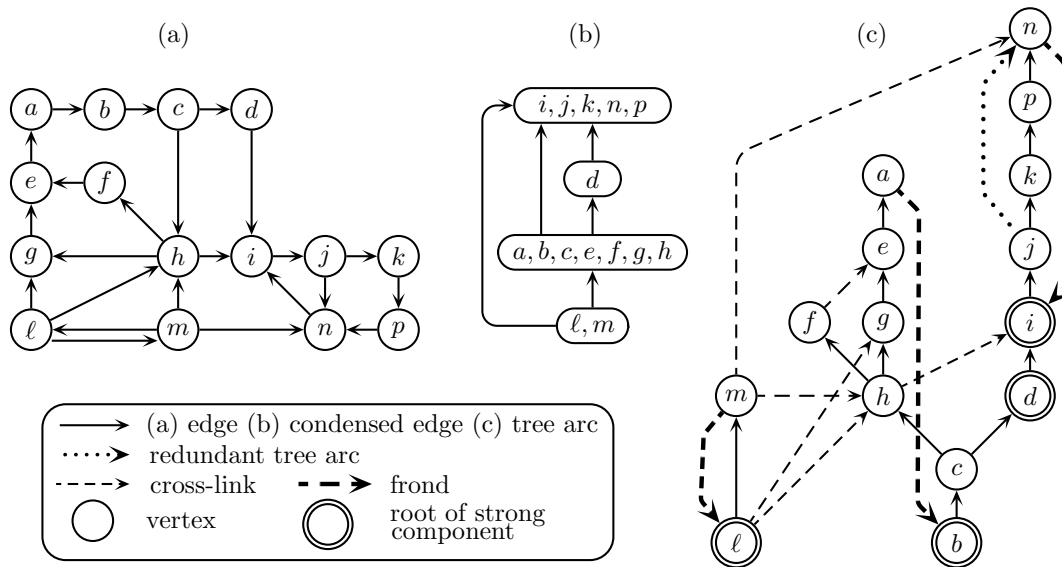


Fig. 8. The running example for this Appendix. (a) A directed graph with 15 vertices and 23 edges. (b) The condensation graph, with 4 strong components. (c) The jungle, obtained by a depth-first search of the graph started from vertices  $b$  and  $\ell$ . Shown are the roots of the 4 strong components, two spanning trees with a total of 13 tree arcs, 3 cycles identified by the 3 fronds, 6 cross-links, and 1 redundant tree arc. See text for more details.

directed graph is said to be *connected* when the corresponding undirected graph, obtained by removing the direction from every edge, is connected.

A digraph  $G = (V, E)$  is said to be *strongly connected* when  $G$  is connected and, for any pair of vertices  $u, v \in V$  there exists a path from  $u$  to  $v$  and a path from  $v$  to  $u$ , i.e., if  $u$  and  $v$  are mutually reachable. Since such a pair of paths is a cycle, a strongly connected digraph can equally well be defined as a digraph where, for any given pair of vertices, there exists a cycle to which both vertices belong. A digraph can be partitioned by partitioning the set of vertices into disjoint subsets. A digraph  $G$  that is connected but not strongly connected can be partitioned into *strong components*. A strong component is a section subgraph of  $G$  which is strongly connected and can not be enlarged without losing this property. Since any given pair of vertices in a strong component belong to some cycle, the definition implies that any given cycle in  $G$  is entirely composed either of vertices in the strong component, or otherwise of vertices of  $G$  not in the strong component. For, if a cycle existed which contained a vertex  $u$  in the strong component and a vertex  $v$  of  $G$  not in the strong component, then the strong component could be enlarged by adding  $v$  to it, in contradiction with the definition. Strong component partitioning is a way to encapsulate cycles, and, as such, very important for the goals of this paper.

A connected digraph  $G$  can be partitioned into a set of vertex-disjoint and edge-disjoint strong components  $C_1, C_2, \dots, C_n$ . If  $(u, v)$  is an edge in  $G$  such that  $u$  is in  $C_1$  and  $v$  is in  $C_2$ , then edge  $(u, v)$  does not belong to any strong component, is an *exit* of  $C_1$  and an *entrance* of  $C_2$ , and *connects*  $C_1$  with  $C_2$ . A strong component can have 0 or more entrances and 0 or more exits.  $G$  must contain at least one strong component with no entrances, and at least one with no exits. A unique *connectivity level structure* can

be obtained by placing all components with no exit in level 1, all components with exits leading to levels 1 through  $i$  in level  $i + 1$ , and finally all components with no entrances in the last level, say  $m$ . More formally:

$$L_1 = \{C_i \mid C_i \text{ has no exit}\} \tag{42}$$

$$L_\ell = \{C_i \mid \text{if } (u, v) \text{ is an exit from } C_i \text{ and } v \in C_j, \\ \text{then } C_j \in L_k, k < \ell\}, \ell = 2, 3, \dots, m.$$

The graph where the strong components are the vertices and their exits and entrances are the edges is called the *condensation graph* of  $G$ . A *dual level structure* and the corresponding condensation can be obtained by starting with all components that have no entrance. The directed graph of Fig. 8(a) has 15 vertices and 23 edges. Vertex  $p$  can be reached from  $a$  but  $a$  can not be reached from  $p$ . The digraph is connected but not strongly connected. It has 4 strong components. Fig. 8(b) shows the corresponding condensation graph and the lists of vertices contained in each condensed vertex.

A *directed tree* is a directed graph where, for a certain vertex  $r$  called the *root*, there is exactly one directed path from  $r$  to any other vertex  $v$  of the tree. The directed tree is and acyclic digraph in which all edges lead away from the root. If a vertex  $u$  happens to be on the path from  $r$  to  $v$ , then  $u$  is an *ancestor* of  $v$  and  $v$  is a *descendant* of  $u$ . The *pedigree* of a vertex is the set of its ancestors, and the *offspring* is the set of its descendants. If  $G$  is a directed graph, a *spanning tree* is a directed tree which contains every vertex of  $G$ . Not every digraph has a spanning tree. However, if  $G$  is a strong component, then every vertex of  $G$  is reachable from any other vertex of  $G$ . Any arbitrary vertex in  $G$  can be designated as the root and a spanning tree can always be found.

### A. Depth-first search of a digraph

*Depth-first search* is a procedure by which all vertices and edges of a graph are visited, starting from an arbitrary vertex, say  $s_1$ , and trying to follow the sequence (vertex – edge – vertex – edge ...) while possible, or *backtracking* to the preceding vertex when the current one lacks unvisited edges. In a directed graph  $G$ , the search can proceed along an edge  $(u, v)$  only in the direction from vertex  $u$  to vertex  $v$ , and can backtrack only from  $v$  to  $u$ . The search identifies the strong components and their roots, creates a forest of spanning trees, classifies edges into 4 categories, and finds properties essential for MMC theory. The search algorithm was given by R. E. Tarjan [18] for a general directed graph. It is briefly reviewed here, with the help of Fig. 8. As the search proceeds, and a certain vertex  $u$  is reached either by the search or by backtracking, a search for unvisited edges leaving  $u$  is performed. When an edge  $(u, v)$  is found, the following 4 cases can arise:

- Vertex  $v$  has never been visited before, and therefore is not yet numbered. Then edge  $(u, v)$  is classified as a *tree arc*,  $v$  becomes the current vertex, and the search continues.
- Vertex  $v$  has been visited and is an ancestor of  $u$ . Then the edge  $(u, v)$  is a *frond* or *back-edge*. Since a path from  $v$  to  $u$  already exists, formed by the tree arcs which connect  $v$  with its descendant  $u$ , the effect of the frond is to close a cycle. Thus,  $v$ , and its pedigree up to and including  $u$ , belong to the same strong component. Vertex  $u$  remains current.
- Vertex  $v$  has been visited and is a descendant of  $u$ . Then the edge  $(u, v)$  is a *redundant tree arc*. A redundant tree arc connects two vertices that are already connected by a path of tree arcs. Such an edge does not affect the strong components. Vertex  $u$  remains current.
- Vertex  $v$  has been visited but is neither an ancestor nor a descendant of  $u$ . Edge  $(u, v)$  is a *cross-link*. Vertices  $u$  and  $v$  may belong to the same or to different strong components. Vertex  $u$  remains current.

The search finds all vertices in  $G$  reachable from the starting vertex  $s_1$ . The tree arcs form a directed tree rooted at  $s_1$  which spans exactly the strong component that contains  $s_1$ , say  $C_1$ , and all other strong components that can be reached from  $C_1$  in the condensation graph, but none other. To find the remaining strong components, the search is restarted from another arbitrarily selected unvisited vertex, say  $s_2$ , and the process is continued until no more unvisited vertices remain. The final result is a *jungle*, which consists of a *forest* with one or more directed trees, each tree spanning exactly one or more strong components, the fronds and the cycles they form, each cycle entirely contained in one strong component, and the cross-links and redundant tree arcs. Fronds serve to detect cycles, and cycles to detect strong components. Cycles can partially overlap, and a given tree arc can belong to more than one cycle. The vertex where the search first enters a strong component is called the *root* of that component, and is also the root of the tree that spans that component and its descendants. In an acyclic digraph, every vertex is a strong component, and every tree in the forest consists of a single vertex. Fig. 8(c) shows the jungle corresponding to the graph in Fig. 8(a).

### ACKNOWLEDGMENT

I am very grateful to Dr. Peter Thieberger of BNL for his unwavering support and companionship, and for reading and extensively commenting what he called this “monumental” manuscript. I am very grateful to Dr. Liwen Shih of UHCL for her interest and youthful enthusiasm, and for the many useful suggestions she made. I am also very grateful to my wife Aurora, for all her support and patience, and I do promise that I will take a vacation.

### REFERENCES

- [1] S. Pissanetzky, “A relational virtual machine for program evolution,” in *Proc. 2007 Int. Conf. on Software Engineering Research and Practice*, Las Vegas, vol. I, pp. 144-150. In this publication, the model was introduced with the name Relational Model of Computation, but was later renamed as the Matrix Model of Computation because of a name conflict.
- [2] S. Pissanetzky, “The matrix model of computation,” in *Proc. 12th World Multi-Conference on Systemics, Cybernetics, and Informatics*, Orlando, FL, 2008, vol. IV, pp. 184-189.
- [3] S. Pissanetzky, “Applications of the matrix model of computation,” in *Proc. 12th World Multi-Conference on Systemics, Cybernetics, and Informatics*, Orlando, FL, 2008, vol. IV, pp. 190-195.
- [4] S. Pissanetzky, “A new type of structured artificial neural networks based on the matrix model of computation,” in *Proc. 2008 Int. Conf. on Artificial Intelligence*, Las Vegas, vol. I, pp. 251-257.
- [5] S. Pissanetzky, “If intelligence is the ability to solve unanticipated problems, then AI needs universal representations,” *Proc. Workshop on Automation and Robotics*, NASA Johnson Space Center, Houston, Texas, 2008.
- [6] For basic material on sparse matrices and an introduction to graph theory for directed graphs, see S. Pissanetzky, *Sparse Matrix Technology*. London: Academic Press, 1984. Russian translation: Moscow: MIR, 1988. Electronic Edition (in English), 2008.
- [7] S. Mahadevan, *Representation Discovery using Harmonic Analysis*. R. J. Brachman and T. G. Dietterich, Ed. San Rafael, CA: 2008.
- [8] W. F. Opydyke, “Refactoring object-oriented frameworks,” Ph.D. thesis, Dep. Comp. Sc., Univ. of Illinois, Urbana-Champaign, 1992.
- [9] G. Wilson, “Refactoring the law: reformulating legal ontologies,” *Juris Dr. Writing Requirement*, School of Law, Univ. of San Francisco, 2006.
- [10] L. Y. Chan, S. Kosuri, and D. Endy, “Refactoring bacteriophage T7,” *Molecular Systems Biology*, Article number: 2005.0018. Published online: September 13, 2005
- [11] A. Garrido and R. Johnson, “Challenges of refactoring C programs,” in *Proc. Int. Workshop on Principles of Software Evolution*, Orlando, FL, 2002, pp. 6-14.
- [12] B. Chandrasekaran, J. R. Josephson, and V. R. Benjamins, “What are ontologies, and why do we need them?” *IEEE Intelligent Systems*, vol. 14(1), pp. 20-26, 1999.
- [13] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design” *IEEE Trans on Software Engng.*, vol. 22, pp. 476-493, 1994.
- [14] C. J. Date and H. Darwen, *Databases, Types, and the Relational Model*. Reading, MA, Addison-Wesley, 2007.
- [15] A. A. Ezhov and D. Ventura, “Quantum neural networks,” in *Future directions for intelligent systems and information sciences*, N. K. Kasabov, Ed. Heidelberg: Physica-Verlag, 2000, pp. 213-235,
- [16] S. Demeyer et al., “The LAN-simulation: a refactoring teaching example,” in *Proc. 8th. Int. Workshop on Principles of Software. Evolution*, Lisbon, 2005, pp. 123-134.
- [17] The code and other teaching materials can be downloaded as free software. Available: [www.lore.ua.ac.be/Research/Artefacts/refactoringLabSession](http://www.lore.ua.ac.be/Research/Artefacts/refactoringLabSession), under the GNU General Public License. See also a motivating example [19].
- [18] R. E. Tarjan, “Depth first search and linear graph algorithms.” *SIAM J. Comput.*, vol. 1, pp. 146-160, 1972.
- [19] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen, “Refactoring-aware configuration management for object-oriented programs,” in *Proc. 29th Int. Conf. on Software Engineering*, Minneapolis, MN, 2007, pp. 427-436.