A Model of Computation for Source Code Analysis and Transformations

Sergio Pissanetzky, Research Scientist. Member, IEEE.

Abstract – Detailed information needed by algorithms that operate on source code is hidden in the code and hard to find. To support the algorithms, various ad hoc code models have been proposed, but the resulting tools are still limited and have language dependencies and poor interoperability. Some contain bugs, perhaps a consequence of the difficulty in developing a tool and a model simultaneously. To address the problem, we propose the Relational Model of Computation (RMC). The RMC consists of two sparse matrices, each representing a set of relations where each tuple is in turn an RMC submodel of finer granularity. The model is a virtual machine, a relational database, and a container for code at the same time. The concept originated in mathematical notation. The RMC is strongly typed and can model OO entities and concepts such as user types, subtypes, single and multiple inheritance for types and behavior, classes of objects, methods, and overloading and overriding. Polymorphism can be described as conditional logic in one of the matrices. Using the hierarchy of submodels, the RMC can model program entities at any detail, including UML designs, functions, three-address code, statements, tokens, even architecture or machine instructions should the need arise. Partial models are possible for local algorithms such as refactoring. In this paper, we formally define the RMC, prove its Turing completeness, discuss its modeling capabilities and possible applications, and illustrate the mechanics of modeling and analysis with examples.

Index Terms – Virtual machines, program representation and analysis, coding tools and techniques, object-oriented design, software maintenance and evolution, refactoring.

1 Introduction

Algorithms that transform or analyze source code treat code as data and attempt to modify or measure its structure. One of the main problems with such algorithms is that details they need are hidden in the code, particularly if it is object-oriented (OO). In response, many algorithms define their own ad hoc representations of the code, a practice that may limit their interoperability and range of applications, and force language dependencies. The code itself remains the only common feature shared by the algorithms.

To address this issue, a *container* for code is needed. A container is an implementation of a standard structure, such as a graph or a database, that supports algorithms designed for that structure, provides common ground, and allows the algorithms to work efficiently and interoperate [1]. The container should be based on a *model* of enough generality. A model is an abstraction, a regular representation of a complex system where irrelevant details are excluded but important details are easily accessible. A model constitutes a mapping from our *viewpoints* or perceptions of the world – such as a business, for example – to some formal entities that represent the viewpoints and allow us to proceed on formal grounds. A relational database, for example, is based on the relational model of data [2].

The use of a container is unavoidable. If developers working on some business problem fail to use a standard container and try to write their own code, they immediately write the code for that container. Programs can be found with several different implementations of the same standard graph structure, or list structure, intercalated with the user code, and none of them fully functional. A similar situation happens with code. Code is data used by the computer to decide what to execute. If the structure of code is extracted, then algorithms can be designed that use the structure to operate in a code-independent manner. The lack of a container for code can influence the development of programming languages and transformation tools.

The concept of a container for code can not be explained without a reference to the concept of *structure*. Structure arises naturally when *regularities* are found in an otherwise disorganized set of data. Structure identifies elements present in the data, reveals their relationships, reduces the overall complexity, and makes the data more manageable and understandable. A nested hierarchy of structures arises because an extracted regularity is in turn data, and thus itself amenable to a further extraction of regularities. Since a level of structure encompasses, hopefully, many cases of data, higher levels become progressively more and more general. In computation, structure is used by the algorithms that operate on the data in a data-independent manner. OO developers are very familiar with the concept. For example, algorithms that operate on lists or queues assume that data is organized as lists or queues.

The concept of creating structure by extracting regularities is a tenet in the theory of complex systems. A program is a complex system. Software development is a good example for the extraction of regularities and the emergence of structural hierarchies, and OO languages, UML diagrams and control flow graphs are examples of tools that help in the process.

But languages, graphs or diagrams are not containers.

We present here the Relational Model of Computation (RMC), and prove that it is a Turing-equivalent virtual machine. Thus, the RMC can emulate any executable algorithm. The model contains a structure of types and embraces the concept of regularization via *submodelling*, a mechanism that closely emulates the nested hierarchical behavior of structure. Typing and submodelling allow the RMC to represent code at any adjustable level of detail, from three-address code to OO code to class diagrams, including machine instructions should the need arise, and even mix them all together for better interoperability. As a result, algorithms have complete control over *granularity*, the level of detail that they need to be aware of, and the ability to hide fine-grained details that need to stay out of the way. The ability to emulate general algorithms, the ability to represent structural hierarchies, the ability to control granularity, the commonality and interoperability attained for many different code representations, qualify the model as a container for code

Logically, the RMC is a relational database represented as a 2-tuple of sparse matrices. The model is simple and easy to comprehend. It is not human-readable, but it is very visual for small examples. This helps with algorithm design, and for presentations and publications such as this one. The RMC is inspired on mathematical notation.

Mathematical notation is mature and precise. For centuries, mathematicians have been writing equations, condensing properties and behavior into objects, restructuring their equations to make them more meaningful and easier to manipulate, using patterns, dealing with complexity, and perfecting a notation that would allow them to express all that. For example, a symmetric positive definite matrix A is an object, and its factorization $\mathsf{A} = \mathsf{LU}$ into the product of a lower triangular matrix L and an upper triangular matrix U, is a pattern, frequently used when solving large systems of linear equations. Mathematical notation consists of equations interspersed with textual descriptions of conditional logic that specify the cases where the equations apply and the *sequences* in which they should be "executed". The notation can be viewed as a language intended for mathematicians. Mathematical notation can describe any algorithm, and, in a sense, it is "Turing-complete". We seek to inherit this property. Powerful transformations, such as the symbolic rationalization of expressions, factorization, symmetrization, or simplification, are common in computer algebra systems [3, 4], and automated bug-free tools are available, including an open framework for symbolic computation within C++ [5]. Links between code transformations and algebraic transformations have been established in the literature [6, 7]. It is only fit that we continue to draw from all that experience.

Fortran, the first major computer language, was created as a *formula translating* system, with statements that formalize conditional logic, cases and sequences, or deal with the subscripts used in matrices and tensors, and variables and operators that represent the equations themselves. Concepts and notations from Fortran were adopted by C, Basic and Pascal, and later by modern OO languages such as C++ and Java. Restructuring code has been practiced even in the early days of Fortran, but, not surprisingly, it intensified and was recognized as the separate discipline of *refactoring* only after the introduction of objects. The structural elements and concepts from mathematical notation, combined in ingenious and meaningful ways, are still found in OO languages.

It is important to point out the differences between our approach and *Structured Programming* [8]. Structured programming also organizes program structures into hierarchies, but that is a property of structure in general. Structured programming is not a container, a database, or a matrix, as we will show the RMC to be. It is a model of programming, but not of computation. And we have no quarrel with the `go to` statement.

1.1 Related Work. Refactoring

We have separated the related work into the areas of *refactoring* and *virtual machines*. Virtual machines are included because the RMC is a virtual machine. Refactoring is representative of analysis and transformations. The RMC is more general than that, but in this paper we restrict our attention to transformations, particularly refactoring because of our interest and because research on refactoring is very active. The following brief survey is partial and intended only as support for arguments used in this paper. To simplify presentation, work is separated into several broad categories.

Refactoring has been recognized as an activity crucial for evolutionary-development processes [9, 10], and the need to teach refactoring in Computer Science curricula has been recognized [11]. Refactoring was originally defined by W. Opdyke as a behavior-preserving program restructuring operation that supports the design, evolution and reuse of OO applications [12]. A catalog of *elementary refactorings* listing more than 70 operations was published by Martin Fowler [13], and a survey of software refactoring was presented [14].

Many methods support manual operations such as finding segments of code that need improvement or deciding which refactorings to apply. In this group, we should mention the application of design patterns to automate refactoring in Java [15], the feature decomposition of programs, where a feature is an increment in program functionality [6], and the detection of program invariants [16].

Methods intended for automation are usually based on some representation of the program other than source code. A graph representation was introduced and graph transformation and rewriting techniques were formalized and used to analyze dependencies in refactoring [17, 18]. A language for refactoring that manipulates a graph representation of the program was proposed [19]. Some approaches advocate automation only at the design level of granularity, where tools are easier to develop, leaving details to the developers. Reducing developer intervention reduces cost. One such approach [20] defines a *space* of alternative designs generated by repeated applications of a refactoring tool, and evaluates them using a design quality function chosen as a combination of multiple metric values such as the CK suite discussed below. The span of the space is obviously limited by the ability of the tool to generate the designs. The application is Java-specific, and falls within the discipline of Search-Based Software Engineering (SBSE).

The use of code metrics is commanding attention. Metrics must satisfy certain mathematical properties in order to be meaningful. The CK metrics suite [21, 22] proposed for OO design is seminal work. It uses 6 different code measurements, such as cohesion and coupling, and is theoretically based on the ontology of Mario Bunge. Other methods use distance cohesion metrics to associate methods and variables of a class [23], a combination of code metrics and visualization [24], and OO metrics for bad smell detection [23]. Mens et al. [14] suggest that refactorings can be classified according to the code quality attributes they affect, such as robustness, extensibility, reusability, or performance, and in terms of attributes such as code size, complexity, coupling and cohesion, which can be measured if access to program structure information is available. Performance can be improved by replacing conditional logic with polymorphism [25].

There has been work on more specialized refactoring systems such as architectural refactoring of inherited software, particularly corporate software [26] and clustering techniques for architecture recovery [27]. And in aspect-oriented programming, using program slicing to improve method and aspect extraction [28], role-based refactoring [29], role mapping [30], and aspect refactoring [31]. Refactoring of non-OO code has also been considered [32]. Code refactoring has been linked to algebraic factoring [6], and, in the particular case of user interface design, to matrix algebra [7].

A *clone* arises when developers copy and paste fragments of code with the purpose of duplicating functionality. Clones are found even in large packages [33, 34] and sophisticated code such as STL [35]. Their presence complicates maintenance and their detection and removal produces more structured code [36]. Clones can be detected by a tool that matches code fragments, but the refactoring actions may be risky [37] and should be decided by developers based on clone differences. One approach [38] uses a comparison algorithm to align tokens and attempts to make the differences more meaningful by interpreting them in terms of programming language entities. Metrics have been used to automatically characterize clones [39]. A proposed method [36] uses Abstract Syntax Trees (AST) to detect exact and near miss clones and removes them by producing in-lined procedures or preprocessor macros in C programs. Another approach [40] proposes a token-by-token comparison and a tool that can extract clones in C, C++, Java, COBOL, and others. Another method uses similar subgraphs in program dependence graphs to find similar code [41], and considers not only the syntactic structure but also the data flow. A lightweight approach based on simple string-matching was proposed [42]. We note that the multiple container implementations we mentioned above are not clones and can not be detected by matching algorithms. Developers often don't realize they are duplicating a container and tend to write fragments of container code where calls to a standard container interface should have been placed.

Pre-processed languages such as C and C++ present special challenges, but not all problems introduced by preprocessing in large OO applications can be handled [43]. Preprocessors were considered from an abstract point of view [44]. In the case of C, enhancements to analysis and refactoring tools [32] are needed to handle preprocessor directives. A toolchest named CScout based on token equivalence classes in C and C++ was proposed [45]. That publication also proposes a relational database with the source code's semantic information that isolates the analysis engine from refactoring transformations. The replacement macro language Astec [46] that can handle preprocessor directives naturally was proposed for refactoring and analysis of C code.

The use of language-based methods to support program evolution has been proposed. Ward and Bennett [47] suggested using the formal intermediate language WSL for evolving legacy code by means of semantic-preserving transformations. The code is translated into WSL, transformations are performed, and the result is translated back into the same or a different language. One problem is that translations can not be formally proved to be correct ([14]). However, such translations are commonplace nowadays for languages such as Java and C#. The macro language Astec was proposed [46] for C programs.

Case studies on refactoring are rare in the literature. A recent Eclipse case study [9] has concluded that even state-of-the-art IDEs such as Eclipse support only a subset of low-level refactorings but lack support for more complex ones, which are also frequent. The study also draws conclusions on high-level design requirements for a refactoring-based development environment. One study [19] notices that existing tools are language-specific and contain significant bugs, even in sophisticated development environments, and proposes a scripting language. Another study reports possible errors [15]. Yet another case study [10] is concerned with designing software repository mining methods for examining the refactoring practice, and with ways for developing insights into the history of the refactorings actually applied by developers. Case studies involving clones have been conducted [40] with JDK, FreeBSD, NetBSD, Linux, and others.

This brief survey shows a disproportion between the considerable research investment and the achievements, and lack of commonality between the tools. Refactoring remains a complex and risky procedure. Many tools are insufficiently generalized, dependent on implementation technology, use ad hoc models, and can not interoperate. The goal is uncertain. Will it satisfy the users? Will there be a tool capable of converting multiple implementations of the graph container to STL containers? No tool can be better than the software model it relies upon. It is difficult to write a tool and at the same time develop an ad hoc model for it. This survey has mentioned the following different models: various types of graphs (AST, CFG, program dependence, for rewriting, for iso-structural subgraph detection), token-parsed code (for token-to-token comparison), token equivalence classes (for preprocessed languages), string-parsed code (for string matching), and various languages and scripts. The RMC proposed in this paper can serve as a container for code and provide commonality for the tools. Existing tools can be ported to the RMC. In Section 4.3 we suggest one way to do it.

1.2 Related Work. Virtual Machines

A virtual machine (VM) [48] is an implementation of an abstract machine such as the Turing machine. An essential feature of all VMs is their ability to emulate a Turing machine, or a finite automaton. VMs provide an interface between programs and operating systems and support program transformations at the intermediate representation level for compiler optimization and performance improvement. A number of VMs have been developed. These include the well-known Java VM for Java, the Common Language Runtime for C#, VB, J# and managed C++, the P-Code VM for Pascal, the Warren VM for Prolog, and the Squeak VM for Smalltalk. The TrueType VM supports the rendering of TrueType fonts and the Sqlite VM supports the operation codes of the Sqlite database system. The Low Level VM (LLVM) was designed for the compile-time and run-time language-independent optimization of programs.

A few VMs serve other purposes. Π-calculus [49], a mathematical formalism used for the analysis of concurrent computation such as communicating systems where the configuration changes with time, is Turing complete [50] and therefore a VM. Spi-calculus [51] is an extension of π-calculus for security applications. VMs are also used in some forms of Abstract Interpretation, a formal mathematical method for static software analysis. To our knowledge, a VM intended for source code transformations has not been defined. Therefore, the impact of VMs on development and code evolution has been limited.

1.3 Organization of this paper

Section 2 introduces the Relational Model of Computation (RMC) as a 2-tuple of matrix representations of sets of relations. It explains how relations are represented as matrices, proposes two conversion algorithms, defines service, sequence, and the matrices of services and sequences, and gives the process algorithm for the RMC. Turing equivalence is proved in Section 3. The proof is by construction. A model of the Turing machine is defined, and then proved to emulate the machine. Section 4 examines important properties such as submodelling, commutativity of sequences and conditionals, and OO features, relationships with other models, possible applications, and a suggestion for an interface for tool support. Finally, Section 5 presents examples to illustrate the mechanics of the model.

2 The Relational Model

The relational model (RMC) describes the structure of a program in terms of relations. In the context of this paper, the model serves as a container for source code and supports algorithms that work with it. The formal definition of the RMC is covered in this section. The C expression

```
a = b - c;                                                          (a)
```

is a description of the *structure* of a function, the table of subtraction, not the function itself. In words: "a is the *codomain*, b and c are the *arguments*, in that order, '-' is the *operator*, and '(a)' is the *name*." The quoted statement describes a tuple in a relation of degree 5. The C statement

```
a = a + 1;                                                          (b)
```

describes the structure of a *mutator*, where "a is the codomain and first argument, '1' is a literal, '+' is the operator, and '(b)' is the name." This quoted statement describes a tuple in a relation of degree 4. In both cases, the tuples describe the *role* played by the variables in the expression. A code fragment such as:

```
if(b)a = 1; else a = 2;                                             (c)
```

would require two tuples in a relation where one of the domains is the domain of *values* of the control variable b, not its role. In the RMC, expressions such as (a) or (b) become *services*, while conditional logic such as (c) becomes a *sequence*. The Relational Model is formally defined as a 2-tuple:

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

where Q is the *matrix of sequences*, and C is the *matrix of services*. Each matrix represents a set of *relations*. In general, both matrices are sparse[1], and the relations have different degrees.

Execution in the RMC proceeds by executing the services in a certain sequence. After execution of a service, logic is invoked to determine what service to jump to, either conditionally or unconditionally. Conditional jumps depend on the values of certain *control variables*. The services are represented in matrix C, and the logic in matrix Q. Thus, execution alternates between the two matrices. Since services and logic are described by relations, execution is equivalent to a search operation in a relational database.

To explain the model, we must explain how a matrix can represent a set of relations, define what is a service and what is a sequence, define a relational representation for both, and give an algorithm for the search. This is the subject of the present section.

To validate the model and qualify it as a virtual machine, we must prove that it can support algorithms by proving its equivalence with the Turing machine. This is covered in Section 3.

To explain the purpose of the model and illustrate its usefulness, we must explain its features and present some examples, which we do in Sections 4 and 5.

A note on notation. Modern relational theory [53] establishes a difference between the definition of *domain* or *type* and the definition of set: a domain is a set with a name. In this presentation, we do not use domain names. Instead, we identify domains by the symbol we use to refer to them, or by a primary key such as an ordinal number, and we use all three terms "set", "domain" and "type" interchangeably, but in context. We also equate sub-type with subset, particularly in the context of type inheritance, but we do not use the terms "subclass" or "superclass." We also use the classic definition of the term "relation" [2]: given n sets $S_1, S_2, \ldots, S_n$, not necessarily distinct, a relation on these n sets is a set of n-tuples each of which has its first element from S_1, its second element from S_2, and so on. S_i is said to be the i-th domain of the relation, and n is its degree. Typically, the relation also has a name and a heading with the names of the domains.

2.1 Representing relations in matrix form

The RMC uses relations to represent and transform data structures, such as the structure of source code, program designs, and other entities. A matrix representation of the set of relations helps to

[1]For a background in sparse matrices see [52]

visualize and comprehend the structure and the transformations. To define the representation, we assume that the given relations are all different (for example, they can have different names), the tuples are all different (for example, they have a unique primary key, or contain foreign keys into a list of relation names), and the domains participating in any given relation are all different. However, a domain is allowed to participate in more than one relation. Let R be the set of the n given relations, irrespective of degree:

$$R = \{r \mid r \text{ is a given relation}\} \qquad n = \mid R \mid. \tag{2}$$

For each $r \in R$ consider the set T_r of its tuples:

$$\begin{array}{l} T_r = \{t \mid t \text{ is a tuple in } r\} \\ T_r \cap T_s = \emptyset \text{ if } s \neq r, \end{array} \tag{3}$$

the set D_r of its domains:

$$\begin{array}{l} D_r = \{d \mid d \text{ is a domain in } r\} \\ D_r \cap D_s \text{ may be } \neq \emptyset \end{array} \tag{4}$$

and define the set T of all tuples, the set D of all domains, and their respective cardinalities:

$$\begin{array}{ll} T = \bigcup_{r\in R} T_r, & k = |T| = \sum_{r\in R} |T_r| \\ D = \bigcup_{r\in R} D_r, & m = |D| \leq \sum_{r\in R} |D_r| \end{array} \tag{5}$$

The matrix H representing R is of dimension $k \times m$. There is a one-one mapping between rows and tuples in T, and between columns and domains in D. Since T is partitioned by relation, a partitioning by relation is also induced in the set of rows. However, since the mapping between D and R is many-many, the mapping between the set of columns and R is also many-many. The following algorithm determines the elements of matrix H:

Algorithm 1. Convert relations to matrix.

```
Step 1.1 For each $r \in R$, $t \in T_r$, $d \in D_r$, determine the value $v_{rtd}$ of the corresponding element of $r$.
Step 1.2 Using the mappings $T \to$ row and $D \to$ column determine the row $i$ corresponding to $t$ and
         the column $j$ corresponding to $d$.
Step 1.3 Set $H_{ij} = v_{rtd}$.
```

The elements not determined by the algorithm are set to null or left blank. Typically the matrix is sparse because only a few elements per row are non-null. Some examples of conversion are presented in Section 2.3, and additional examples can be found in Section 5. The following algorithm determines R from a given matrix H:

Algorithm 2. Convert matrix to relations.

```
Step 2.1 For each row $i$ in $\mathsf{H}$, use the mapping row $\to T$ to determine the corresponding tuple $t \in T$.
Step 2.2 Use the mapping $T \to R$ to determine the corresponding relation $r \in R$.
Step 2.3 For each column $j$ in $\mathsf{H}$, if $H_{ij}$ is not null, use the mapping column $\to D$ to determine the
         corresponding domain $d \in D$.
Step 2.4 Find domain $d$ and tuple $t$ in relation $r$, and set $v_{rtd} = H_{ij}$.
```

2.2 Variables

A *variable* is an entity with three properties, a *name*, a *role*, and a *value*. If a variable is named v, the role is designated as $v.\rho$ or ρ_v, and the value as $v.\nu$ or ν_v. The corresponding *role domain* or domain of roles is $v.R$, where $v.\rho \in v.R$, and the *value domain* or domain of values is $v.V$, where $v.\nu \in v.V$. The value is the traditional value of a variable, and the domain of values is the traditional *type* or user type of the variable. The domain of roles is the same for all variables. For a variable v, the domain of roles is the set of *role identifiers*:

$$v.R = \{a_1, a_2, \ldots, c_1, c_2, \ldots, m_1, m_2, \ldots\} \tag{6}$$

where a, c, m stand for argument, codomain, and mutator, respectively, and the ordinal subscripts are explained below. The role domain for a literal such as 1 is:

$$1.R = \{a_1, a_2, \ldots\} \tag{7}$$

because literals can only be arguments. A *control variable* is one that is used in the model to control conditional logic. For example, if a program consisted of the C expressions (a), (b) and (c) of Section 2, then $V = \{a, b, c\}$ is the set of all variables, and $V' = \{b\}$ is the set of all control variables, because b is a control variable. Below, we discuss why the condition $V' \subseteq V$ must always be satisfied.

2.3 Services

At the heart of the RMC is the concept of *service.* A service s is a mapping that involves the value domains of a set V_s of k_s distinct variables:

$$V_s = \{v \mid v \text{ is a variable in } s\}, \quad k_s = |V_s|. \tag{8}$$

Depending on the roles played by the variables in the service, set V_s is partitioned into the subsets of *arguments, codomains* and *mutators.* Let A_s, M_s and C_s be the cartesian products of the value domains of the arguments, mutators and codomains, respectively. Formally, the service is

$$s \;:\; A_s \;\times\; M_s \;\rightarrow\; M_s \;\times\; C_s. \tag{9}$$

Our aim is to use relations to represent services by describing the roles played by the variables, that is, the *structure* of the service, not the mapping itself. This approach effectively separates the structure in a data-independent manner, and leads to a formulation that allows structure to be algebraically manipulated. With this in mind, we assign a role identifier to each variable in s, chosen from the set of Equation (6) in such a way that a, c or m identify the variable as an argument, codomain or mutator, and the ordinal number corresponds with the order that the variable appears in the service declaration. The relation that represents the structure of service (9) is defined as follows:

> The relation is of degree $k_s + 1$ and contains one single tuple. A domain O contains the name of the service. The remaining k_s domains are the role domains of the variables $v \in V_s$, contain the roles identifiers of the variables, and are labeled accordingly.

We say that a service is *executed* when given values are provided for the argument and mutator variables, and some process determines the corresponding values for the mutator and codomain variables. Implicit in the definition of execution is that assignments have been made to arguments and mutators, and upon return, assignments are made to mutators and codomains.

Figure 1 illustrates several examples. Column K contains a primary key for identification purposes, and the example expressions are listed in the second column. The variables used in these services are v_1, v_2, v_3, v_4. For brevity, we refer to their value domains as $\nu_1, \nu_2, \nu_3, \nu_4$. The remaining columns show the corresponding tuples in tabular form, and are labeled with "O" and the names of the roles. Line 1 is an assignment, where v_1 is the codomain and v_2 is the argument. Lines 2 and 3 are in infix operational notation, where o is the operator. When the service is provided by an operator, the operator itself defines the roles. In line 2, v_1 is the codomain and v_2, v_3 are the arguments, but in line 3, v_1 is a mutator and v_2 is the argument. Line 4 represents a function f that returns a value but does not change its arguments (*const* arguments in the C++ sense). Line 5 is an example of conversion of an expression directly from C, which describes the mapping $+\!=\!: \nu_2 \times \nu_1 \rightarrow \nu_1$. Line 6 is the transition

K	Expression	O	$v_1.\rho$	$v_2.\rho$	$v_3.\rho$	$v_4.\rho$
1	$v_1 \leftarrow v_2$	$=$	c_1	a_2		
2	$v_1 \leftarrow v_2 \; o \; v_3$	o	c_1	a_2	a_3	
3	$v_1 \leftarrow v_1 \; o \; v_2$	o	m_1	a_2		
4	$v_1 \leftarrow f(v_2, v_3, v_4)$	f	c_1	a_2	a_3	a_4
5	$v_1 \mathrel{+}= v_2;$	$+=$	m_2	a_1		
6	$\delta : \nu_1 \times \nu_2 \to \nu_1 \times \nu_2 \times \nu_3$	δ	m_1	m_2	c_3	

Figure 1: Service examples.

function δ of a Turing machine in set notation, where the value domains ν_1, ν_2, ν_3 are the states, the alphabet, and set $\{\ell, r\}$, respectively.

If the column labeled "Expression" is eliminated from Figure 1, what remains is exactly a *matrix of services*, as formally defined below. Section 5 contains additional examples. Services are used to construct relational models. A service declaration is an interface, implemented by a fundamental operation such as lines 1, 2, 3 or 5 of Figure 1, by three-address code, by a function such as line 4, or by another program, a component, an Internet service, a library subroutine, an OS function. The implementation of a service can be described by another RMC, and a RMC can be described as a service, and therefore as part of another RMC. Conversely, suitable sections of the RMC can be isolated, represented by a service, and extracted as a separate submodel. This process simulates the extraction of regularities, mentioned in Section 1. Services and RMCs are interchangeable. It is this interchangeability what gives the RMC its power and versatility.

To define the matrix of services C, consider a set of services, and let m be its cardinality:

$$S = \{s \mid s \text{ is a service}\}, \quad m = |S| \tag{10}$$

For each $s \in S$, let V_s be the corresponding set of variables, Equation (8), and define the set V of all variables used in the services of set S:

$$V = \bigcup_{s \in S} V_s, \qquad n = |V| \le \sum_{s \in S} |V_s| \tag{11}$$

In general, services share variables and sets V_s are not disjoint. The following steps define the matrix of services C:

(1) Represent each service $s \in S$ as a relation. This creates a set of single-tuple relations of various degrees.
(2) Represent the set of relations in matrix format (Section 2.1), with a single domain O containing the names of the services.
(3) Add a domain K for primary keys.

The resulting matrix of size $m \times (n + 2)$ is the matrix of services for set S. This is in fact, the same procedure we followed to create the table in Figure 1. Some or all of the domains in Equation 9 can be empty, with the definition adjusted accordingly. Non-empty domains in (11) can be omitted from the matrix of services if they are of no interest for the analysis at hand. For example, details of a service "advance tape" are of no interest, unless of course we are designing a tape recorder.

2.4 Sequences

Sequences are used to organize the order of execution of the services. After a service has finished execution, a jump is used to determine which service will execute next. The jump is implemented as a relation, called a *sequence relation.* The RMC accepts two branching decision mechanisms: unconditional jumps, and conditional jumps. Unconditional jumps have one single destination and are described by a sequence relation with a single tuple. Conditional jumps depend on the values of one or more discrete-valued *control variables.* They have multiple destinations, and are described by a sequence relation with multiple tuples.

All sequence relations contain the domains P and F ("previous" and "following"). Both P and F contain foreign keys into the domain K of matrix C, which contains the primary keys that uniquely identify the services. The keys in P identify a service that has just executed, the keys in F identify the possible destinations of the jump. If the jump is unconditional, no other domains are necessary. For a conditional jump, consider a service $s \in S$ and the conditional logic that makes the branching decision after service s has executed. Define the set V'_s of control variables:

$$\begin{aligned} V'_s &= \{v \mid v \text{ is a control variable in the conditional logic}\} \\ n'_s &= \mid V'_s \mid \end{aligned} \tag{12}$$

We have $V'_s \subseteq V_s$, because values for the variables in V'_s must be initialized or calculated prior to use, and that requires a service. For an unconditional jump $V'_s = \emptyset, n'_s = 0$. Let n_c be the total number of possible distinct combinations for the values of the control variables. The sequence relation that represents the conditional logic following service s is defined as follows:

> The relation is of degree $n'_s + 2$, and contains n_c tuples, one in correspondence with each combination of control values. In each tuple, domain P contains a foreign key into domain K of matrix C that identifies service s. Domain F contains a foreign key into domain K that identifies the destination service for that combination. The remaining n'_s domains contain the corresponding values of the control variables in V'_s, and are labeled accordingly.

Figure 2 illustrates a few examples of sequence relations. The examples are based on a simplified matrix of services, also shown in the figure. The flow control logic is in column "code", written in C-like code. There are two examples. The first example is a conditional jump. After service $s1$ is executed, control jumps to either service $s2$ or $s3$, depending on the value of the control variable v. The sequence relation has 2 tuples, one for each possible value of v. Both tuples contain 1 in domain P, because 1 is the primary key for service $s1$. The keys in domain F indicate the jumps. The last tuple corresponds to the unconditional jump to $s3$ after $s2$ has finished execution.

The second example shows a `switch` statement, also invoked after service $s1$ has executed. The first relation has 2 tuples, one for each possible value a or b of the switch variable w. There are two more sequence relations with one tuple each, describing the unconditional jumps to service $s4$ after services $s2$ and $s3$ have completed execution, respectively. Additional examples can be found in Section 5.

$$\mathsf{C} = \left|\begin{array}{cc} \text{K} & \text{service} \\ 1 & s1 \\ 2 & s2 \\ 3 & s3 \\ 4 & s4 \end{array}\right|$$

code	P	$v.\nu$	$w.\nu$	F
$s1;$ if$(v)\{s2;\}s3;$	1	true		2
	1	false		3
	2			3
$s1;$ switch $(w)\{$	1		a	2
case a : $s2;$ break;	1		b	3
case b : $s3;$ break; $\}$	2			4
$s4;$	3			4

Figure 2: Sequence examples, based on the simplified matrix of services C

If the column labeled "code" is eliminated from Figure 2, what remains is a matrix of sequences (except for the domain of actors A, discussed below). The matrix of sequences Q, Equation (1), is the matrix representation of the set of sequence relations for all services. Let S be the set of services, Equation

(10), let $s \in S$ be a service, consider the control variables V'_s, Equation (12), and define the set V' of all control variables:

$$V' = \bigcup_{s\in S} V'_s, \quad n' = |V'| \le \sum_{s\in S} |V'_s| \tag{13}$$

In general, services share variables and sets V'_s are not disjoint.

In addition, *actors* must be considered. Actors initiate sequences. An actor initiates a sequence by activating an unconditional jump to a service. There can be many actors and many different sequences in a model. The actors are not necessarily independent. A typical case is where an actor that starts from a menu depends on another that displays the menu. Sequences end in a special *exit* service. There is no jump after exit, and execution can resume only if an actor activates another sequence. To construct matrix Q follow these steps:

(1) For each service s except exit, consider the flow control logic immediately following service s.
(2) Convert the logic to a relation, as discussed above in this section. Since $V' \subseteq V$, use the same subscripts to identify variables in V' as were used to identify the corresponding variables in V.
(3) Represent the set of sequence relations in matrix format (Section 2.1).
(4) Add a domain A for actors, and tuples representing unconditional jumps to the corresponding services. The result is matrix Q.

Once again, structure has been extracted in a data-independent manner. This time, it is the structure of the execution sequences, extracted to matrix Q and separated from the data, the values of the control variables. The decision of where to jump to is always made in matrix Q. Matrix Q retains detailed control over the sequences.

2.5 The process algorithm

The process algorithm is the algorithm that executes the RMC. This algorithm, already explained in Section 2, is formally defined here. We assume that matrices C and Q defined in Equation (1) are given, and we use dot notation to refer to domains in the matrices, for example $\mathsf{Q}.P$ would be domain P in matrix Q. The searches mentioned in the algorithm can be implemented as relational select and join operations. The definition of V'_s given in Equation (12) is used in the algorithm.

Algorithm 3. Relational Model Process. For any actor $a \in \mathsf{Q}.A$:

```
Step 3.1 (Enter execution) Search $\mathsf{Q}.A$ for the tuple that matches $a$ and find the foreign key $f$ in $\mathsf{Q}.F$
         for that tuple.
Step 3.2 (Find service) Search $\mathsf{C}.K$ for the match to $f$ and find service $s$ in $\mathsf{C}.O$.
Step 3.3 (Stop on exit) If $s =$ exit, stop. Otherwise, continue.
Step 3.4 (Execute a service) Execute service $s$.
Step 3.5 (Find sequence relation) Search matrix $\mathsf{Q}$ for the set $\Theta$ of all tuples that satisfy $\mathsf{Q}.P = f$.
Step 3.6 (Find control variables) Using the set $\Theta$ just found, determine $V'_s$ for service $s$.
Step 3.7 (Find jump) Search $\Theta$ for the tuple that satisfies $(\forall v \in V'_s)\, \mathsf{Q}.v.\nu = v.\nu$
Step 3.8 (Follow jump) Find a new value for $f$ in $\mathsf{Q}.F$ in the resulting tuple and go to Step 3.2.
```

All matches must be unique, assuming the conditional logic is correct. This completes the definition of the relational model. We will now prove the equivalence of the RMC with the simple Turing machine to qualify the RMC as a virtual machine.

3 Turing Equivalence

In this section we prove that the RMC is equivalent to a simple Turing machine. By proving it, and since the Turing machine is considered equivalent to execution of a program, we will have proved that the RMC is also equivalent to execution of a program and qualifies as a virtual machine. The Turing machine can be defined as a 7-tuple [54]:

$$\mathcal{T} = (\Omega, \Sigma, \Gamma, \delta, \omega_s, \omega_a, \omega_r) \tag{14}$$

where Ω is the (finite) set of states, Σ is the (finite) input alphabet and $\flat \notin \Sigma$, Γ is the (finite) tape alphabet, $\flat \in \Gamma$, $\Sigma \subset \Gamma$, δ is the transition function, $\delta : \Omega \times \Gamma \to \Omega \times \Gamma \times \Lambda$, and $\omega_s, \omega_a, \omega_r \in \Omega, \omega_r \neq \omega_a$ are the start, accept and reject states, respectively, $\flat$ indicates a space or blank, and $\Lambda = \{\ell, r\}$. The process algorithm for the Turing machine is:

Algorithm 4. Turing Machine Process.

```
Step 4.1  (Initialize) Initially, the input string $w \in \Sigma^*$ is in the leftmost cells of the tape, and the rest
          of the tape is blank. The head is in the leftmost cell and the machine is in state $\omega_s$.
Step 4.2  (Read) Read symbol from cell under the head.
Step 4.3  (Transition) Execute $\delta$ to find the new state, the new symbol, and the direction.
Step 4.4  (State) Go to the new state.
Step 4.5  (Test) If the new state is $\omega_a$ or $\omega_r$, stop. Otherwise continue.
Step 4.6  (Write) Write the new symbol to the cell under the head.
Step 4.7  (Move) If the head is on the leftmost cell and the direction is left, go to Step 4.2. Otherwise
          move one step in the indicated direction.
Step 4.8  (Loop) Go to Step 4.2.
```

where "*" is the star operation or Kleene closure. The theorem is as follows:

Theorem. *Every Turing machine has an equivalent relational model.* □

The proof is by construction.We first construct a relational model of the Turing machine, and then prove that the model emulates the Turing machine.

3.1 The Relational Model of the Turing Machine

To model the Turing machine, we associate a variable of type Γ with each cell, and let G be the set of all these variables. We further introduce the following variables:

g	the array of cell variables, a variable of type {array of variables of type Γ}.
g^0	the input string, a variable of type {array of variables of type Σ}.
γ	the current cell value, of type G.
c	the current cell, of type {cell}.
c^0	the initial cell, of type {cell}.
ω	the current state, of type Ω.
λ	the head direction, of type Λ.
b	true if $\omega = \omega_a, \omega_r$ of type {true, false}.

We recall that a type is a set and note that {array ...} is notation for a set of arrays. An array is an ordered set. Define set V of Equation (11) as follows:

$$V = \{g, g^0, \gamma, c, c^0, \omega, \omega_s, \omega_a, \omega_r, \lambda, b\} \tag{15}$$

where we have used ω_s, ω_a and ω_r as variables of type Ω. Define matrices C and Q as follows:

$$\mathsf{C} = \begin{array}{c|cccccccccccc|} O & c & \omega & \gamma & \lambda & g & b & g^0 & \omega_s & \omega_a & \omega_r & c^0 \\ \hline \texttt{in} & c_1 & c_2 & & & c_3 & & a_4 & a_5 & & & a_6 \\ \texttt{rd} & a_1 & & c_3 & & a_2 & & & & & & \\ \texttt{tr} & & m_1 & m_2 & c_3 & & & & & & & \\ \texttt{lg} & & a_1 & & & & c_4 & & & a_2 & a_3 & \\ \texttt{wr} & a_1 & & a_2 & & c_3 & & & & & & \\ \texttt{mv} & m_1 & & & a_2 & & & & & & & a_3 \\ \texttt{ex} & & & & & & & & & & & \end{array} \qquad \mathsf{Q} = \begin{array}{|cccc|} A & P & b & F \\ \hline 1 & & & \texttt{in} \\ & \texttt{in} & & \texttt{rd} \\ & \texttt{rd} & & \texttt{tr} \\ & \texttt{tr} & & \texttt{lg} \\ & \texttt{lg} & \text{true} & \texttt{ex} \\ & \texttt{lg} & \text{false} & \texttt{wr} \\ & \texttt{wr} & & \texttt{mv} \\ & \texttt{mv} & & \texttt{rd} \end{array} \qquad (16)$$

To simplify notation, we have omitted domain K of primary keys from matrix C, and refer to the services by their names. We have also labeled some columns with the names of the corresponding variables, although, of course, we refer to the role domains in matrix C, and to the value domains in matrix Q. There is only one actor, say actor 1. Initially, all cell variables in set g are set to $\flat$ (blank). Given are: $g^0, \omega_s, \omega_a, \omega_r$, and c_0, the leftmost cell in the tape. Service `in` initializes c form c_0 and ω from ω_s, and writes array g^0 left-justified to g. Service `rd` reads a value for γ from cell c in array g. Service `tr` is the Turing transition function. It finds values for ω, γ and λ from given values of ω and γ. Service `lg` calculates boolean variable b to be true if ω is either the accept or the reject state, or false otherwise. Service `wr` writes the value of γ into cell c of array g. Service `mv` moves the head left or right from cell c depending on the values of λ and c^0, and sets a new value for c. It needs c^0 because it must avoid moving left from the leftmost cell. Service `ex` returns control or stops execution.

Note that Ω, Σ and Γ do not appear explicitly in the model. Ω and Γ appear as given constants in the model of the transition function δ, and Σ appears as a given constant in the model of the input string g^0. These two models, however, are irrelevant for our purposes. Note also that b appears in matrix C only as a codomain, but seemingly is never used. This is because b is a control variable, and is used in matrix Q to control the flow control logic.

3.2 Turing Equivalence of the RMC

Two machines are equivalent if they recognize the same language. If a language is recognized by a Turing machine, we must show the existence of a relational model that recognizes it. We do that by converting the Turing machine into an equivalent RMC that emulates the Turing machine. In fact, we have converted the Turing machine defined in Equation (14) into the RMC defined by matrices C and Q of Equation (16). We must only show that this RMC simulates the Turing machine.

Execution of the RMC process algorithm, Algorithm 3, actor 1, on matrices C and Q, simulates step-by-step the execution of Algorithm 4, the Turing process algorithm. Algorithm 3 executes the services in matrix C in the following order:

(1) Service `in`. This simulates Step 4.1.
(2) Service `rd`, which simulates Step 4.2.
(3) Service `tr`, simulating Steps 4.3 and 4.4.
(4) Service `lg`, the calculation of control variable b used by the RMC.
(5) Either service `ex`, a simulation of the first part of 4.5, or service `wr`, a simulation of the second part of 4.5 and 4.6.
(6) Service `mv`, which simulates Step 4.7.
(7) Loop to Service `rd`, simulating Step 4.8. ■

The theorem proves that there is a RMC for every computation that can be executed on a Turing machine, and qualifies the RMC as a virtual machine. The significance of this conclusion is that matrices C and Q, with the relations and domains they contain, given by Equation (1), formally define the structure of an executable program. The elements of the matrices are the data of the program. The separation between structure and data allows algorithms to be designed that use the structure to operate on the data in a data-independent manner and can interoperate among each other.

We said in the Introduction that the relational model of computation is inspired in mathematical notation. The ability of mathematical notation to describe all algorithms is inherited by the RMC, and makes it a virtual machine, a universal container for all executable algorithms.

4 Properties and Applications of the Relational Model

The RMC was defined by Equation 1. The preceding sections have covered related definitions and fundamental properties of the RMC. In this section we introduce additional terminology and examine other important features and techniques of the RMC that can be applied for designing algorithms. The mechanics of operation of the RMC is illustrated by the examples in Section 5.

4.1 Size and scalability

A crude estimate of the size of a full RMC model hierarchy can be made as follows. Roughly speaking, there would be one submodel per class for an OO program, or one submodel per function or subroutine for a non-OO program. The main model consists of a list of services, one per each method in the program, each service occupying one line in matrix C. Method or function declarations go to the main model, executable statements from the functions or methods go to the service matrices of the corresponding submodels, and conditional statements and polymorphic class hierarchy declarations go to the sequence matrices of the submodels[2]. The model and submodels have about the same granularity as the source code. Rows in the service matrices only contain role identifiers (see, e.g., Figure 1), considerably shorter than names used in code. Rows in matrix Q should also be about the same size or smaller than the corresponding conditional statements in the code. Thus, the size of the model, measured in characters, should be smaller than or comparable to the size of the source code.

Some applications may require a finer granularity. Example E1 in Section 5 illustrates a problem of refactoring where a section of code has been refined all the way down to three-address code. This should not affect the overall size, however, because refactoring is usually a local procedure.

Scalability is achieved in mathematical notation by defining concepts of progressively greater generality. Scientists use these concepts in their reasoning, while keeping in mind the more detailed definitions. The corresponding technique in the RMC is *submodelling*, where parts of a model are removed and described by a submodel, leaving the main model smaller and more compact and manageable. submodelling improves both performance and scalability, and submodels can nest to any depth.

The power of relational databases to handle large volumes of information of all kinds is well-known. The power of sparse matrices to handle very large problems is also well-known. Engineers routinely solve problems with matrices having millions of rows and columns. A sparse matrix with a constant number of non-null elements per column grows in size only linearly with the size of the problem. This is the case for the matrix of services for the low-coupling, high-cohesion case. The growth may be somewhat faster for a high-coupling case.

The combination of submodelling, relational databases and sparse matrices holds a promise for excellent performance and scalability. Yet, the fact that an algorithm is supported does not mean it will operate properly. Each algorithm must be investigated case by case. The most promising algorithms are the ones that take advantage of the structure, properties and techniques of the RMC.

[2]A reviewer has pointed out that converting polymorphism to conditional logic can cause scalability problems, and has suggested a static points-to analysis [55] to reduce the complexity of the required conditionals.

4.2 Model extensions

Using the RMC as-is may not be practical in all cases, and additions may be needed when implementing specific algorithms. The RMC is a database, a set of relations coupled with primary key/foreign key links. Extra relations can always be added and linked to represent the additional information, without upsetting the general operation or properties. We refer to such additions as *extensions*. Extensions are used to represent program features not included in the formal definition. For example, an encapsulation algorithm that creates classes and methods, may have to keep track of method-to-class assignments. To do it, simply add a column to matrix C with class names in correspondence with the methods. Inheritance relationships for methods can be dealt with in a similar manner. An extra row in C can keep track of the organization of a hierarchy of classes or a structure of types and subtypes. In all cases, however, it would probably be better to use separate relations in order to keep the database better normalized, but this is an implementation detail.

To add a relation to an existing matrix simply modify sets T and D as needed and add the new rows and columns to the existing matrix. Merging two matrices, or removing a relation or a subset of relations from a matrix, are both straightforward. These features support the property of *model nesting* of the RMC.

4.3 Relationships with graphs and other models

To every model or submodel in the RMC hierarchy, irrespective of granularity, there corresponds a labeled directed graph $G = (V, E)$. The vertices include the actors in matrix Q and the services in matrix C, labeled with the roles played by the parameters in the services. The edges are the tuples in matrix Q – each of which contains the foreign keys for two distinct vertices – appropriately labeled with the values of the control variables existing in the tuple. Directed graph concepts such as paths and cycles, strong components, section subgraphs, directed adjacency level structures, and others, directly apply to each submodel of the RMC. A wealth of powerful existing graph algorithms such as breadth-first and depth-first search, subgraphing and graph rewriting also apply. What can be done with the graph can also be done with the relations. In this paper, sometimes we use graph notation and terminology when there is no equivalent relational terminology.

For a submodel of fine granularity, where the services are similar to three-address code, the corresponding graph is similar to the well-known *control flow graph* (CFG) of a program[56]. Considerable experience gained with CFGs for manipulating programs can be directly applied to the RMC. Well-known procedures used to form the CFG from the program could be used to form the RMC submodel. An Abstract Syntax Tree (AST), a Data Flow Diagram (DFD), or a three-address code version of the program, can all be similarly obtained from a fine-grained RMC submodel, and similar considerations apply.

The direct conversion of UML models into RMC models is a possibility. We have been able to (manually) convert static structure diagrams such as class diagrams with little effort. Most features map directly, others such as member visibilities require additional tuples. A UML-RMC collaboration is possible. The RMC is a relational database, and as such it is ideally suited to handle detailed OO information. The RMC can also be created from business rules, and an example is given in Section 5.3.

It is well known that a database can handle a very large volume of information, and that front end interfaces can be designed to retrieve that information and present it to users in many different forms. Similarly, an interface can be designed that hides the RMC and emulates the functionality of other structures, such as the ones mentioned near the end of Section 1.1, or even UML diagrams or source code. Such an interface would not be very efficient, but it would allow easy porting of existing tools and testing of new ones.

4.4 Commutativity and permutation of services and sequences

Let c_{sd} be a non-null element in matrix C, corresponding to service $s \in S$ and domain $d \in D$, and let φ_{sd} be the first symbol in the value of c_{sd}. Thus, the value of φ_{sd} is either a, c, or m. If $\varphi_{sd} = c$, service s is said to be a *domain constructor* for domain d. A domain can have many constructors, but must have at least one. A service can be a domain constructor for any number of domains, or none.

Consider two different services s and s'. Services s and s' are said to *commute* if they satisfy the following two conditions:
(1) s and s' are consecutively and unconditionally sequenced. In other words, matrix Q contains one and only one tuple that has a foreign key for s' in domain F, and this tuple contains no conditional logic and has a foreign key for s in domain P. In CFG terminology, we would say that s is an *immediate dominator* of s'.
(2) For every domain $d \in D$, either (2a) φ_{sd}, or $\varphi_{s'd}$, or both, are null, or (2b) $\varphi_{sd} = \varphi_{s'd} = a$. Domain-disjoint services always satisfy (2a). We note that a pair (a, a) in the column of domain d means that the value of the corresponding variable is being used by both services s and s', but not changed, so the order doesn't matter. Any other combination would involve c or m and would mean that the value is changed by at least one of the services, in which case order matters.

The order of execution of two commutative services can be reversed without affecting the behavior of the program. Service commutation is a matrix operation that affects matrix Q, not C. A *permutation* of the sequence of services is obtained when commutation is applied one or more times. A permutation is said to be *behavior-preserving* if it can be obtained as a sequence of commutations between services that commute. A behavior-preserving permutation amounts to a behavior-preserving refactoring of the program.

An *execution path* in the matrix of sequences Q corresponds to a directed path in the control flow graph. The path consists of a sequence of tuples linked together. A *cycle* is a closed path. For example, in matrix Q of Equation (16), the sequence of tuples (`rd-tr`, `tr-lg`, `lg`-true-`ex`) is an execution path, and (`rd-tr`, `tr-lg`, `lg`-false-`wr`, `wr-mv`, `mv-rd`) is a cycle. Under certain conditions, execution paths are independent and can be executed in any order. A typical case is a *fork*, where conditional logic selects among two or more possible paths in such a way that only one of them is followed at a time. These concepts are well known in control flow graph theory, and will not be expanded here.

Domains in a relation always commute. The notation we use is designed to preserve this property, because arguments, mutators and codomains are qualified with ordinals making their order in the service declaration independent from any order assumed for the domains. The columns of matrices C and Q can be permuted in any arbitrary order.

There is a form of commutativity between control variables that may have an important effect on refactoring when single-inherited polymorphism (SIP) or multiple-inherited polymorphism (MIP) are used for implementation. It applies to control flow branches that depend on two or more conditionals. Consider the following C programs:

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

Programs A, B and C are equivalent, but many different implementations are possible. They must be examined in the light of the fact that the value domains of control variables in matrix Q are not ordered and always commute. There exists an inherent fundamental symmetry between variables such as a or b in Program A. This symmetry is preserved if Program A is implemented using MIP. If there are n_a possible values of a, and n_b of b, a total of $n_a + n_b$ base classes are required. An upgrade to a new value of a, for example, would require just one new class. A symmetry-preserving SIP implementation is also possible if a and b are combined into a composite domain (with $n_a n_b$ values), but this would require $n_a n_b$ different base classes, and an upgrade to a new value of a would require n_b new classes.

This may be prohibitive if $n_a, n_b > 2$.

The forms B, C, instead, discriminate one condition at a time, forcing an artificial order between the control variables and breaking the symmetry. The result is a very large number of choices, none of them very good. Consider a general case with n control variables with 2 possible values each. A simple analysis demonstrates that there are $n!$ different possible orderings among the n variables. If either SIP or conditionals are used for implementation, there are 2^{2^n-1} possible combinations for each ordering, for a grand total of $n!\,2^{2^n-1}$. For example, there are 16 ways of coding a link with 2 control variables, and 768 for one with 3 control variables, and this is just one link. Designers may not even realize they are dealing with such multiplicity and leave it for developers to sort it out. This is a good example of the support that the RMC can provide for software analysis. Example E2 of Section 5.3 expands on the matter.

4.5 Modeling object-oriented features

The RMC models OO features naturally. The columns of matrix C represent types. A partition of the set of columns into subsets models type encapsulation and creation of user types, with each subset representing a new user type. Column subpartitions create type hierarchies that can model single and multiple type inheritance [53].

The rows of matrix C represent services. A partition of the set of rows into subsets models service encapsulation and the creation of methods, with each subset representing a new method. Row partitions must be compatible with the sequences of execution, and an appropriate permutation of the sequences must exist for the encapsulation to be legal. Conditions under which behavior-preserving permutations of the sequences of execution are possible have been discussed in Section 4.4.

Classes of objects can be modelled as associations between methods and user types, and method inheritance naturally combines with type inheritance. A class can in turn be converted into a separate service and described by a separate submodel, thus effectively eliminating it from matrix C and creating recurrence for model processing.

Method inheritance is also modelled naturally. In the traditional OO sense, a derived class inherits all the attributes and methods from its ancestors, and can contain additional attributes and methods and overrides of the inherited methods. This definition makes the attributes and methods of a derived class a *superset* of the attributes and methods of the parents. To reduce confusion, we avoid terms such as subclass or superclass, and use terms such as derived, base, ancestor or descendant for classes, and subset or superset for sets. A derived class corresponds to a superset. Consider, for example, the set D of all domains, Equation (5), and let D_1 be a subset of D and D_2 a subset of D_1, so that $D_2 \subseteq D_1 \subseteq D$. Then D_1 contains or "inherits" all the types in D_2, and, if D_2 is a proper subset of D_1, then D_1 has some additional types of its own. Similarly, a method that uses domains in D_2, also uses domains in D_1. If D_1 and D_2 become classes, then D_2 is the base and its superset D_1 is the derived class, and if the method is assigned to D_2, it is "inherited" by D_1. Multiple inheritance works in the same way. Let $D_3 \subset D_1$ be another subset of D_1. Then D_1 is a superset of both D_2 and D_3, and its class is derived from both.

4.6 Submodelling

Submodelling is a very important technique. Under certain conditions, a subset of domains can be converted into a composite domain and a subset of services into a composite service, and described by a separate submodel. The submodel can be *extracted* from the main model, leaving only the declarations for the composite service and the composite domain. A submodel can also be *inserted* into the main model. Extracting a submodel makes the main model smaller and more amenable for high level transformations. Inserting a submodel refines the granularity and allows more detailed transformations. By way of the extraction and insertion of submodels, the granularity of the main model, or parts of it, can be adjusted to the desired level of refinement, as appropriate for the problem at hand. A submodel is a complete model and can, in turn, be submodelled.

Behavior-preserving permutations of the sequences of execution may be needed to meet the required conditions. The purpose of the permutations is to bring certain services together in the sequence. Subsequently, the reordered set of services is partitioned, and the subsets are encapsulated to create higher level services or methods.

A partition of the set of domains is also necessary, but domain permutations are not needed because domains are not ordered. The set of domains is partitioned, and the subsets are encapsulated to create composite domains. All permutations and partitions are logical, and no movement of data is needed. Finally, the composite services and domains are extracted into submodels, and service and domain declarations are inserted in their place. Example E1 of Section 5.1 illustrates submodelling.

4.7 An RMC-centric development environment

Traditionally, the source code is the main repository or "source" of a program. However, OO technology is intended to facilitate developer-program communication, and by its very nature, it is limited in its ability to serve as data for algorithms. The fact that OO code is not a good container for such algorithms has led to the development of many different code representations reported in the literature, some of which we discussed in Section 1.1. Instead, an RMC-centric environment can be considered, where the RMC is the formal repository and the code remains as a means for program-developer communication. Other authors have considered similar ideas. O'Keeffe *et al.* [20] believe that the goal of the OO approach is to minimize the cognitive complexity of programming tasks. A *refactoring-based* development environment has been proposed [9]. A relational database with the source code's semantic information that effectively isolates the analysis engine from the refactoring transformations and metric extraction was proposed as well [45]. The feasibility of the idea depends on the ability of the RMC to automatically communicate with code and other models. Separating the two functions – a communication hub and a formal repository – can prove advantageous for both.

5 Examples

The examples are intended to illustrate the mechanics of operation of the RMC and the support that the RMC can provide to emulate software analysis and transformation algorithms. The RMC is designed to be operated by an algorithm. The model does not make decisions. Instead, it provides an environment for algorithms to perform analysis tasks and automate the decisions. In this paper we discuss the environment, not the algorithms, but we must emphasize the model's most useful features and support for the algorithms.

The model is not intended for the developers to view. Instead, developers would most likely see a menu of options, prepared by an algorithm after performing a suitable analysis of the model. Developers may have access to "outside information", such as business rules, an existing design that calls for a particular class structure, or knowledge of a good solution for a similar problem. The menus should allow them to use the information and guide the algorithm, and to enter names of their choice so the results are not cryptic. The algorithm may also propose a choice of viable solutions, rated according to some metrics. If this is a refactoring run from existing software, developers may want to choose combinations that "blend" better with the rest of the code. The model precludes illegal transformations and promotes uniform style and good programming practices.

The model will be small in most cases. For example, when refactoring, developers would typically select a section of code that needs improvement and direct the algorithm to refine that section. The algorithm can, then, find other related sections and refine them as well. The process is not unlike to what developers do when they refactor manually, and should be amenable to automation because the model has the required information.

In the rare cases where nothing at all is known about the program, the process would be one of analysis. The RMC can find some promising permutations, examine them, rate them based on some design metrics, find applicable patterns, and present the results as suggestions to the developers. For an OO program, CK metrics is appropriate. CK metric consists of a suite of 6 different measurements. Suppose the RMC can select some legal encapsulations as candidates and obtain values for Couplings Between Objects (CBO) and Lack of Cohesion in Methods (LCOM) for the objects and methods that would result from each candidate. The developers would be able to select one of the options and produce code with the desired properties, such as robustness and understandability, which improve with high cohesion and low coupling, or high coupling for better performance.

In this case, where nothing at all is known about the program, the number of possible unqualified permutations may be very large, hence the selection of promising permutations is an important issue. The process we are discussing is one of regularization, akin to the extraction of regularities from a complex body of information with the purpose of reducing its complexity (Section 1). The more "regular" the regularities are and the more frequently they appear, the more "promising" the permutation is. Example E1 suggests some possibilities. Our practical experience indicates that the number of attractive choices is usually very limited. Maintainers also know that there aren't too many good refactorings for a given program.

The Turing machine model of Section 3.1 serves as an example as well. It illustrates direct modeling from business rules, submodelling, the use of granularity, the use of conditionals, and the ability of the RMC to mix granularities. Services `in` and `lg` in matrix `C` are very detailed, while service `tr`, the transition function, is coarse-grained. Several services need submodels. Service `in`, for example, initializes 3 domains at once and can be described by a separate submodel with 3 services, one for each domain, that execute in sequence. In the submodel, the service that initializes domain g, which is an array, also needs a submodel with details on how to initialize an array. For our purposes, however, the submodels are all irrelevant. An example of the use of conditionals is provided by the tuples (`lg`, true, `ex`) and (`lg`, false, `wr`) in matrix `Q` of the Turing model. We note that we have constructed this model directly from the definition or "business rules" of the Turing machine. We feel that this possibility is important and should be explored further.

5.1 Example E1. Part I.

Example E1, Part I illustrates the following:

- the mechanics of the relational model;
- modeling a straight-line fragment of code using a very fine granularity;
- commutativity and row and column permutations;
- encapsulation by matrix partitioning and alternative encapsulations;
- formation of classes and methods;
- model nesting and submodelling techniques;
- the refactorings Move Method and Rename Method [13], and other Rename refactorings.

In this example we follow Chidamber and Kemerer [21] and apply the notion of *designer viewpoints*, a set of empirical binary relationships between the elements of an OO design that represent the designer's intuitive understanding of complexity. The viewpoints satisfy certain axioms, and map to formal entities, which we can then use to proceed on formal and coherent grounds. This is the essence of the generally accepted concept of *model*, used by engineers throughout history and advocated in this paper.

Of particular interest to us when working with the RMC is the concept os *similarity*. Given two entities with comparable properties, similarity is the intersection of their set of properties, and *degree* *of similarity* is the cardinality of that intersection. For example, if variables are properties of methods, the similarity of two methods is the set of variables they share. Designers mind similarity because it allows them to control important metrics such as *cohesion* and *coupling.* Encapsulating similar methods and their shared variables into an object class makes the class more cohesive. Encapsulating *all* pairwise similar methods makes the class less coupled with other classes.

Example E1 illustrates RMC support for these concepts by outlining the mechanics of encapsulation and submodelling. To present the example, we use a simple algorithm that considers similarities between methods and between roles of variables. We caution the reader that there is more to encapsulation than this. For example, depth of inheritance and number of children also affect coupling, but are not covered in the example. Example E1 begins with Program P1, written in C in a style reminiscent of three-address code, an intermediate representation language used by many compilers and virtual machines (see for example [57]). The exercise results in two different object-oriented designs, D2 and D3, and their corresponding C++ programs P2 and P3, both equivalent to Program P1. Since P1 can be obtained from either P2 or P3, conversion between P2 and P3 is also possible, and the example amounts to a refactoring of an OO program. Example E1 has some similarity with an example used in our early work [58]. Program P1 is shown in Figure 3. There is only one actor and no conditional logic. We can omit the matrix of sequences and assume that execution follows the order of the rows in the matrix of services, starting with the first row. The matrix of services can also show the partitions used to encapsulate user types and define classes and methods, and even the refactorings themselves. It is an ideal tool for the presentation of simple examples.

We use upper-case letter as class names, such as G or H, and lower-case letters followed by a number as object names, such as g1, g2, g3 for objects of class G. Methods are named with a single upper-case letter followed by the name of the class they belong to, and a number, e.g. AG1, AG2 are two occurrences of method AG of class G. Individual variables are named with one or two lower-case letters. Matrix C^0, shown in Figure 3, is the matrix of services for Program P1.

```
PROGRAM P1
1. d = 1;
2. a = 2;
3. b = 3;
4. rx = 4;
5. ry = 5;
6. vx = 6;
7. vy = 7;
8. fx = 8;
9. fy = 9;
10. ta = a * fx;
11. tb = a * fy;
12. tc = d * vx;
13. td = d * vy;
14. te = ta + tc;
15. tf = tb + td;
16. tg = b * fx;
17. th = b * fy;
18. rx = rx + te;
19. ry = ry + tf;
20. vx = vx + tg;
21. vy = vy + th;
```

$$\mathsf{C}^0 = \begin{array}{cc|cccccccccccccccccccccccccc|}
\text{K} & \text{O} & \text{d} & \text{a} & \text{b} & \text{rx} & \text{ry} & \text{vx} & \text{vy} & \text{fx} & \text{fy} & \text{ta} & \text{tb} & \text{tc} & \text{td} & \text{te} & \text{tf} & \text{tg} & \text{th} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
1 & = & c & & & & & & & & & & & & & & & & & 1 & & & & & & & & \\
2 & = & & c & & & & & & & & & & & & & & & & & 1 & & & & & & & \\
3 & = & & & c & & & & & & & & & & & & & & & & & 1 & & & & & & \\
4 & = & & & & c & & & & & & & & & & & & & & & & & 1 & & & & & \\
5 & = & & & & & c & & & & & & & & & & & & & & & & & 1 & & & & \\
6 & = & & & & & & c & & & & & & & & & & & & & & & & & 1 & & & \\
7 & = & & & & & & & c & & & & & & & & & & & & & & & & & 1 & & \\
8 & = & & & & & & & & c & & & & & & & & & & & & & & & & & 1 & \\
9 & = & & & & & & & & & c & & & & & & & & & & & & & & & & & 1 \\
10 & \ast & & 1 & & & & & & 2 & & c & & & & & & & & & & & & & & & & \\
11 & \ast & & 1 & & & & & & & 2 & & c & & & & & & & & & & & & & & & \\
12 & \ast & 1 & & & & & 2 & & & & & & c & & & & & & & & & & & & & & \\
13 & \ast & 1 & & & & & & 2 & & & & & & c & & & & & & & & & & & & & \\
14 & + & & & & & & & & & & 1 & & 2 & & c & & & & & & & & & & & & \\
15 & + & & & & & & & & & & & 1 & & 2 & & c & & & & & & & & & & & \\
16 & \ast & & & 1 & & & & & 2 & & & & & & & & c & & & & & & & & & & \\
17 & \ast & & & 1 & & & & & & 2 & & & & & & & & c & & & & & & & & & \\
18 & + & & & & m & & & & & & & & & & 2 & & & & & & & & & & & & \\
19 & + & & & & & m & & & & & & & & & & 2 & & & & & & & & & & & \\
20 & + & & & & & & m & & & & & & & & & & 2 & & & & & & & & & & \\
21 & + & & & & & & & m & & & & & & & & & & 2 & & & & & & & & &
\end{array}$$

Figure 3: Program P1 and the matrix of services C^0 of size 21×28. Two-letter column captions have been stacked to make them fit. For the same reason, we have written "1" or "2" instead of a_1 or a_2, and we use unsubscripted "c, m" to identify codomains and mutators. There should be no confusion.

We will now apply suitable permutations to rows and columns of C^0 and encapsulate services into methods and domains into user types, and associate them to create classes of objects. We assume throughout that nothing is known about the problem, and base our refactoring decisions on similarity alone, as discussed above. Cases where nothing at all is known about a problem are rare. In this case, we may have been told that program P1 describes one time step of a simulation of the motion of a mass particle under the action of a force, in two dimensions. This alone would have prompted us to use the well known *vector* class, and to command the RMC to encapsulate (rx, ry), (vx, vy), and (fx, fy) together. However, as we show below, program P1 has an alternative encapsulation worth considering: (d, a, b). We missed this option ourselves when we first refactored program P1 manually, and noticed it only after we examined matrix C^0 for regularities. It seems likely that developers would have missed it as well, because of the vector preconception, and a comparison between the two options would never have been made. This remark suggests that model-based mechanical analysis may complement human analysis, and justifies our assumption.

If nothing at all is known about program P1, the permutations should be determined by examination of matrix C^0 for regularities. We begin by partitioning away domains K and O, and all the literals, which are of no interest for the encapsulation. Examining the remaining columns, one may notice that some have 2 entries, others have 3. Among the ones with 3 entries, we notice the similarities $\{c, 2, m\}$ for vx and vy, and $\{c, 2, 2\}$ for fx and fy. These 2 pairs are candidates, but they only appear twice each. There are also 3 columns with pairwise similarities $\{c, 1, 1\}$, suggesting the encapsulation of (d, a, b). This is a stronger regularity because it appears 3 times.

The analysis can be extended further by examining the services associated with the candidate domains. For example, the c in domains vx, vy corresponds to services 6 and 7, both of which have a 1 under the literals, which increases the rating of vx, vy as candidates. In the case of (d, a, b), all 6 services associated with the 1's have the pairwise similarity $\{1, 2, c\}$ in their rows, and all 3 services associated with the c's have 1 in their rows. This is a strong similarity. A larger problem would probably have rows and columns with many more entries, and yield even stronger similarities. [3] Proceeding along these lines, and applying only legal permutations consistent with the commutativity rules of Section 4.4, two different solutions were obtained: matrix C^1 of Figure 4 was obtained when starting from (d, a, b), and matrix C^2 of Figure 8 was obtained when starting from (vx, vy) and (fx, fy). Matrix C^2 is discussed in Section 5.2.

Next, we discuss matrix C^1. The 17 domains of interest in matrix C^1 have been partitioned into 3 subsets, g1, h1 and h2. Subsets h1 and h2 have identical structures. Subset g1 is an object of a class, which we shall call G, and subsets h1 and h2 are objects of a class to be called H.

The 21 services are partitioned into 5 subsets: CG1, CH1, CH2, MH1, MH2. Subset CG1 contains the 3 domain constructors for the domains in subset g1, and is therefore the class constructor for class G, hence the name. Subsets CH1 and CH2 have identical structures. They are two appearances of the class constructor for class H. They become method CH. Subsets MH1 and MH2 have identical structures. They are two appearances of a method of class H. They both become method MH.

Domain subsets h1 and h2 are sub-partitioned in two parts each. For h1, these parts are (rx, vx, fx) as class attributes, and (ta, tc, tg, te) as temporaries for method MH1. This is indicated by the corresponding domain constructors. Domain h2, of course, is similarly subpartitioned.

At this point, we can use matrix C^1 to illustrate an application of the Fowler Move Method refactoring. Suppose a request is made (quite improperly) to move method MH1 to class G. One way to achieve that would be to replicate temporaries ta, tc, tg, te as temporaries in class G, and give class G access privileges to modify class H members rx, vx. This is the complete Move Method refactoring. Rename refactorings are also simple. In a typical implementation, all names would be in a separate normalized relation, and the matrices would contain foreign keys into that relation instead of the actual names used in this example. To change the name of a method, class, class attribute, method argument,

[3] One of the reviewers suggested using a search-based approach [20] to automate the decisions.

$$\mathsf{C}^1 = \left|\begin{array}{cc|ccc|ccccccc|ccccccc|ccccccccc|c}
 & & & \text{g1} & & & & & \text{h1} & & & & & & & \text{h2} & & & & & & & & \text{literals} & & & & & \\
\text{K} & \text{O} & \text{d} & \text{a} & \text{b} & \text{rx} & \text{vx} & \text{fx} & \text{ta} & \text{tc} & \text{tg} & \text{te} & \text{ry} & \text{vy} & \text{fy} & \text{tb} & \text{td} & \text{th} & \text{tf} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \\
\hline
1 & = & c & & & & & & & & & & & & & & & & & 1 & & & & & & & & & \\
2 & = & & c & & & & & & & & & & & & & & & & & 1 & & & & & & & & \text{CG1} \\
3 & = & & & c & & & & & & & & & & & & & & & & & 1 & & & & & & & \\
\hline
4 & = & & & & c & & & & & & & & & & & & & & & & & 1 & & & & & & \\
6 & = & & & & & c & & & & & & & & & & & & & & & & & & 1 & & & & \text{CH1} \\
8 & = & & & & & & c & & & & & & & & & & & & & & & & & & & 1 & & \\
\hline
5 & = & & & & & & & & & & & c & & & & & & & & & & & 1 & & & & & \\
7 & = & & & & & & & & & & & & c & & & & & & & & & & & & 1 & & & \text{CH2} \\
9 & = & & & & & & & & & & & & & c & & & & & & & & & & & & & 1 & \\
\hline
12 & * & 1 & & & & 2 & & & c & & & & & & & & & & & & & & & & & & & \\
10 & * & & 1 & & & & 2 & c & & & & & & & & & & & & & & & & & & & & \\
16 & * & & & 1 & & & 2 & & & c & & & & & & & & & & & & & & & & & & \text{MH1} \\
14 & + & & & & & & & 1 & 2 & & c & & & & & & & & & & & & & & & & & \\
18 & + & & & & m & & & & & & 2 & & & & & & & & & & & & & & & & & \\
20 & + & & & & & m & & & & 2 & & & & & & & & & & & & & & & & & & \\
\hline
13 & * & 1 & & & & & & & & & & & 2 & & & c & & & & & & & & & & & & \\
11 & * & & 1 & & & & & & & & & & & 2 & c & & & & & & & & & & & & & \\
17 & * & & & 1 & & & & & & & & & & 2 & & & c & & & & & & & & & & & \text{MH2} \\
15 & + & & & & & & & & & & & & & & 1 & 2 & & c & & & & & & & & & & \\
19 & + & & & & & & & & & & & m & & & & & & 2 & & & & & & & & & & \\
21 & + & & & & & & & & & & & & m & & & & 2 & & & & & & & & & & & \\
\hline
\end{array}\right|$$

Figure 4: The permuted and partitioned matrix C^1. The matrix contains object g1 of class G and two objects of class H. CG1 is the constructor for class G. There are two occurrences of a constructor CH for class H, and two occurrences of a method MH of class H. Object g1 of class G is passed to MH as an input argument. A sub-partition of h1 and h2 separates attributes for class H from temporaries used by method MH.

$$\mathsf{C}_{CG} = \left|\begin{array}{ccccccc}
\text{O} & \text{d} & \text{a} & \text{b} & \text{i1} & \text{i2} & \text{i3} \\
= & c & & & 1 & & \\
= & & c & & & 1 & \\
= & & & c & & & 1
\end{array}\right|
\qquad
\mathsf{C}_{MH} = \left|\begin{array}{ccccccccccc}
\text{O} & \text{d} & \text{a} & \text{b} & \text{r} & \text{v} & \text{f} & \text{t1} & \text{t2} & \text{t3} & \text{t4} \\
* & 1 & & & & 2 & & & c & & \\
* & & 1 & & & & 2 & c & & & \\
* & & & 1 & & & 2 & & & c & \\
+ & & & & & & & 1 & 2 & & c \\
+ & & & & & m & & & & & 2 \\
+ & & & & m & & m & & & 2 &
\end{array}\right|$$

$$\mathsf{C}_{CH} = \left|\begin{array}{ccccccc}
\text{O} & \text{r} & \text{v} & \text{f} & \text{i4} & \text{i5} & \text{i6} \\
= & c & & & 1 & & \\
= & & c & & & 1 & \\
= & & & c & & & 1
\end{array}\right|$$

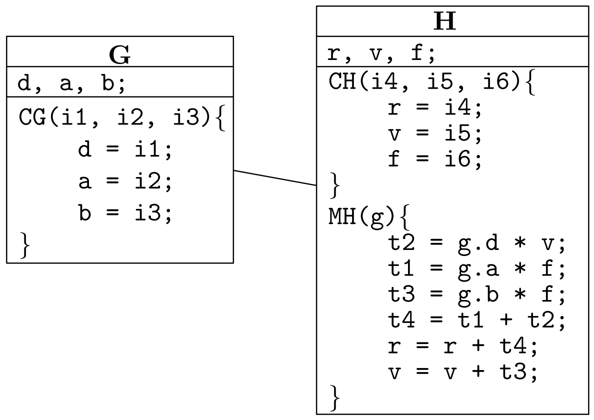
Figure 5: The submodels CG, CH and MH for matrix C^0 of Figure 3. Primary keys and sequences have been omitted.

$$\mathsf{C}^{11} = \left|\begin{array}{ccccccccccccc}
\text{O} & \text{g1} & \text{h1} & \text{h2} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{CG} & c & & & 1 & 2 & 3 & & & & & & \\
\text{CH} & & c & & & & & 1 & & 2 & & 3 & \\
\text{CH} & & & c & & & & & 1 & & 2 & & 3 \\
\text{MH} & 1 & m & & & & & & & & & & \\
\text{MH} & 1 & & m & & & & & & & & &
\end{array}\right|$$

Figure 6: Matrix C^{11}, the simplified main model corresponding to matrix C^0 of Figure 3.

etc, all that is needed is to change that name in one place.

We are now ready to define the submodels. There is one submodel for method CG, one for method CH, and one for method MH. They are shown in Figure 5. When the submodels are extracted from C^1, the main model becomes much smaller. It is shown in Figure 6. The submodels and the reduced



```
G
d, a, b;
CG(i1, i2, i3){
    d = i1;
    a = i2;
    b = i3;
}
```

```
H
r, v, f;
CH(i4, i5, i6){
    r = i4;
    v = i5;
    f = i6;
}
MH(g){
    t2 = g.d * v;
    t1 = g.a * f;
    t3 = g.b * f;
    t4 = t1 + t2;
    r = r + t4;
    v = v + t3;
}
```

```
PROGRAM P2
main{
    G g1(1, 2, 3);
    H h1(4, 6, 8);
    H h2(5, 7, 9);
    h1.MH(g1);
    h2.MH(g1);
}
```

Figure 7: Design D2, the final design for Example E1, Part I. Program P2 is one possible C++ version of Program P1, corresponding to the main model C^{11} of figure 6. The classes G and H correspond to the submodels of Figure 5. g1 is an object of class G, and h1, h2 are objects of class H.

main model are all RMC models on their own right, and are ready for a repeat of the entire process we have just described: row and column permutations, encapsulation of services and domains, definitions of classes and methods, submodelling. We do not pursue this here.

The final design for example E1 can be obtained from matrix C^{11} and the submodels, and is shown in Figure 7. If a language module were available, it would be expected to produce code, for example in C++ or Java, directly from the design. Alternatively, developers can write the code, and then parse it back and compare with the relational model for debugging. The operation is similar to a compiler's, because the code won't "compile" until any bugs introduced by the developers have been fixed, except that the compiler can only enforce the rules of a language while the RMC enforces the formal structure of the program. Program P2, shown in Figure 7, is the C++ code corresponding to Program P1, Part I, manually obtained from Design D2.

5.2 Example E1. Part II.

Example E1, Part II illustrates the following:

- multiplicity of solutions;
- refactoring object-oriented code;

The second part of Example E1 is based on Program P1 and its relational model matrix C^0 of Figure 3, but it uses matrix C^2, which was obtained from matrix C^0 by encapsulating the pairs (vx, vy) and (fx, fy). Matrix C^2 is shown in Figure 8.

Matrix C^2 describes two classes, G and V. Object g1 is an instance of class G, and CG1 is the class constructor for class G. Objects v1, v2, v3, v4, v5, v6 and v7 are all instances of class V. Class V has one constructor with 3 invocations, CV1, CV2, and CV3. Class V also has three methods, method WV with the 3 invocations WV1, WV2, and WV3, method PV with 1 invocation PV1, and method QV with the 2 invocations QV1 and QV2. Class V will immediately be recognized as the very well-known class *vector* in two dimensions. Method WV is an overload of operator '*' for the vector. It takes a double and returns a vector. Method PV is an overload of operator '+', which takes a vector and returns another. And method QV is an overload of operator '+=', which takes a vector and modifies its object. Class *vector* is indeed the accepted standard for program P1. Matrix C^2 corresponds to design D3 (not shown), and to the C++ program Program P3, shown in Figure 8.

Program P2 is shorter than P3, reflecting the fact that similarity $\{d, a, b\}$ is stronger than $\{vx, vy\}$, $\{fx, fy\}$. Both P2 and P3 are C++ refactorings of P1. Since P1 can be obtained from either P2 or

$$
\mathsf{C}^2 = \begin{array}{ccccccccccccccccccccccccccccc}
 & & \multicolumn{3}{c}{g1} & \multicolumn{2}{c}{v1} & \multicolumn{2}{c}{v2} & \multicolumn{2}{c}{v3} & \multicolumn{2}{c}{v4} & \multicolumn{2}{c}{v5} & \multicolumn{2}{c}{v6} & \multicolumn{2}{c}{v7} & \multicolumn{9}{c}{\text{literals}} & \\
\text{K} & \text{O} & \text{d} & \text{a} & \text{b} & \text{r} & \text{r} & \text{v} & \text{v} & \text{f} & \text{f} & \text{t} & \text{t} & \text{t} & \text{t} & \text{t} & \text{t} & \text{t} & \text{t} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \\
 & & & & & \text{x} & \text{y} & \text{x} & \text{y} & \text{x} & \text{y} & \text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} & \text{h} &&&&&&&&&& \\
1 & = & c &&&&&&&&&&&&&&&&& 1 &&&&&&&&& \\
2 & = && c &&&&&&&&&&&&&&&&& 1 &&&&&&&& \text{CG1} \\
3 & = &&& c &&&&&&&&&&&&&&&&& 1 &&&&&&& \\
4 & = &&&& c &&&&&&&&&&&&&&&&& 1 &&&&&& \text{CV1} \\
5 & = &&&&& c &&&&&&&&&&&&&&&&& 1 &&&&& \\
6 & = &&&&&& c &&&&&&&&&&&&&&&&& 1 &&&& \text{CV2} \\
7 & = &&&&&&& c &&&&&&&&&&&&&&&&& 1 &&& \\
8 & = &&&&&&&& c &&&&&&&&&&&&&&&&& 1 && \text{CV3} \\
9 & = &&&&&&&&& c &&&&&&&&&&&&&&&&& 1 & \\
10 & * && 1 &&&&&& 2 && c &&&&&&&&&&&&&&&&& \text{WV1} \\
11 & * && 1 &&&&&&& 2 && c &&&&&&&&&&&&&&&& \\
12 & * & 1 &&&&& 2 &&&&&& c &&&&&&&&&&&&&&& \text{WV2} \\
13 & * & 1 &&&&&& 2 &&&&&& c &&&&&&&&&&&&&& \\
16 & * &&& 1 &&&&& 2 &&&&&&&& c &&&&&&&&&&& \text{WV3} \\
17 & * &&& 1 &&&&&& 2 &&&&&&&& c &&&&&&&&&& \\
14 & + &&&&&&&&&& 1 && 2 && c &&&&&&&&&&&&& \text{PV1} \\
15 & + &&&&&&&&&&& 1 && 2 && c &&&&&&&&&&&& \\
18 & + &&&& m &&&&&&&&&& 2 &&&&&&&&&&&&& \text{QV1} \\
19 & + &&&&& m &&&&&&&&&& 2 &&&&&&&&&&&& \\
20 & + &&&&&& m &&&&&&&&&& 2 &&&&&&&&&&& \text{QV2} \\
21 & + &&&&&&& m &&&&&&&&&& 2 &&&&&&&&&&
\end{array}
$$

```
PROGRAM P3
main{
   G g(1, 2, 3);
   V r(4, 5);
   V v(6, 7);
   V f(8, 9);
   V tab,tcd,tef,tgh;
   tab = f.WV(g.a);
   tcd = v.WV(g.d);
   tgh = f.WV(g.b);
   tef = tab.PV(tcd);
   r.QV(tef);
   v.QV(tgh);
}
```

Figure 8: Matrix C^2, obtained from matrix C^0 of Figure 3 by encapsulating the pairs (vx, vy), (fx, fy). This matrix corresponds to design D3 (not shown). Also shown is the C++ program Program P3, obtained from C^2. This model has two classes, G and V. Class G has a constructor, and class V has a constructor and 3 methods, WV, PV and QV. Program P3 is the C++ version of Program P1 of Example E1, Part II.

P3, the example amounts to a refactoring between P2 and P3. P2 and P3 are very different, and a conversion between the two would be very difficult to accomplish using combinations of elementary refactorings. P2 is better, but P3 is standard for problems of the type of Example E1. The reason is outside of Example E1: the use of vectors becomes necessary when dealing with the rotation of rigid bodies.

5.3 Example E2

This example illustrates the following:

- creating the RMC directly from business rules;
- refactoring an object-oriented design;
- using a very coarse granularity;
- using relational operations to induce refactorings;
- symmetry and commutativity of multiple-conditioned sequences, and
- model support for inheritance and polymorphism.

This is a good example of the kind of analysis that can be carried out with the help of an RMC model. The example emphasizes the ability of the RMC to expose the inner structure of a program and help the designer to make informed decisions. In this case, a source of serious confusion, that appears to have remained unnoticed so far, is revealed. The problem statement is: calculate the cost to outfit an aircraft model A, B or C to carry mail (M) or spray crops (S). A simple high-level analysis of this problem indicates that the following services and control variables are needed:

- services AM, BM, CM determine the list of mail equipment for each of model A, B, or C.
- services AS, BS, CS determine the list of spray equipment for each of model A, B, or C.

- services MCost, SCost calculate the cost of mail or spray equipment.
- the control variables are: a (select M/S) and b (select A/B/C).

The execution algorithm should initialize a and b, call one of AM, BM, CM, AS, BS, or CS to determine the list of equipment, and call one of MCost or SCost to determine the cost, passing the list. This information is enough to create the RMC model, which is shown in Figure 9.

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

Figure 9: The relational model for Example E2, obtained directly from business rules. A relation R of degree 4 with six tuples is marked in matrix Q.

The model is complete, but, before a design can be created, a decision must be made regarding the implementation of the control variables in matrix Q (see discussion in Section 4.4). We would like to produce designs that use combinations of conditional logic and single-inherited polymorphism (SIP). SIP makes decisions based on a single condition. While it is possible to combine a and b and form a single control variable suitable for SIP implementation, we will not explore that possibility. Instead, we will modify matrix Q in such a way that the decisions controlled by a and b are made separately. The process is similar to the transformation of Program A in Section 4.4 into Programs B and C, and just as in Section 4.4, there are two different ways of doing it depending on which decision is made first, a or b.

Let R be the relation of degree 4 indicated in matrix Q of Figure 9. We must transform R into two relations that use a single control variable each. The relational operation that achieves that goal is normalization. Normalization is a well known procedure [2]. R must be normalized. There are two ways of doing it because R is unnormalized in both a and b, and order matters. We start by normalizing domain a first. We note that $a.\nu = \{M, S\}$ and augment R with the new domain $\alpha = \{1, 2\}$ containing foreign keys into a. Then, we project R on $\{P, A, \alpha\}$ and on $\{\alpha, b, F\}$, respectively. The result are relations R1 and R2, which have been substituted for R and shown in Figure 10 as part of the normalized matrix Q_{12}. R2 can still be normalized for b, but this will not be necessary. R1 and R2 contain the same information as R. In fact, a join of R1 and R2 on R1.F = R2.P yields R. The new keys in domain α are in fact two new fictitious "do-nothing" services introduced for convenience.

To normalize in the reverse order, starting with $b.\nu = \{A, B, C\}$, we add the fictitious services $\beta = \{3, 4, 5\}$ as keys for domain b, and project R on $\{P, b, \beta\}$ and $\{\beta, a, F\}$, respectively. The remaining normalization with respect to a is superfluous. The result are relations R3 and R4, which we have inserted into matrix Q_{34} and are identified in Figure 10. A join of R3 and R4 on R3.F = R4.P yields R, as it should.

$$\mathsf{Q}_{12} = \left|\begin{array}{lllll|l} A & P & a & b & F & \\ 1 & & & & \mathtt{init} & \\ \hline & \mathtt{init} & M & & 1 & \\ & \mathtt{init} & S & & 2 & \text{R1} \\ \hline & 1 & & A & \mathtt{AM} & \\ & 1 & & B & \mathtt{BM} & \\ & 1 & & C & \mathtt{CM} & \\ & 2 & & A & \mathtt{AS} & \text{R2} \\ & 2 & & B & \mathtt{BS} & \\ & 2 & & C & \mathtt{CS} & \\ \hline & \mathtt{AM} & & & \mathtt{MCost} & \\ & \mathtt{BM} & & & \mathtt{MCost} & \\ & \mathtt{CM} & & & \mathtt{MCost} & \\ & \mathtt{AS} & & & \mathtt{SCost} & \\ & \mathtt{BS} & & & \mathtt{SCost} & \\ & \mathtt{CS} & & & \mathtt{SCost} & \\ & \mathtt{MCost} & & & \mathtt{exit} & \\ & \mathtt{SCost} & & & \mathtt{exit} & \end{array}\right. \qquad \mathsf{Q}_{34} = \left|\begin{array}{lllll|l} A & P & a & b & F & \\ 1 & & & & \mathtt{init} & \\ \hline & \mathtt{init} & & A & 3 & \\ & \mathtt{init} & & B & 4 & \text{R3} \\ & \mathtt{init} & & C & 5 & \\ \hline & 3 & M & & \mathtt{AM} & \\ & 3 & S & & \mathtt{AS} & \\ & 4 & M & & \mathtt{BM} & \text{R4} \\ & 4 & S & & \mathtt{BS} & \\ & 5 & M & & \mathtt{CM} & \\ & 5 & S & & \mathtt{CS} & \\ \hline & \mathtt{AM} & & & \mathtt{MCost} & \\ & \mathtt{BM} & & & \mathtt{MCost} & \\ & \mathtt{CM} & & & \mathtt{MCost} & \\ & \mathtt{AS} & & & \mathtt{SCost} & \\ & \mathtt{BS} & & & \mathtt{SCost} & \\ & \mathtt{CS} & & & \mathtt{SCost} & \\ & \mathtt{MCost} & & & \mathtt{exit} & \\ & \mathtt{SCost} & & & \mathtt{exit} & \end{array}\right.$$

Figure 10: Two versions of matrix Q prepared for SIP or conditional implementations. Relations R1, R2, R3 and R4, all of degree 3, are identified.

Matrix Q_{12} involves 3 different decisions, one based on values of a and two based on values of b. If combinations of conditionals (C) and polymorphism (P) are used, then there are 8 possible combinations: CCC, CCP, CPC, CPP, PCC, PCP, PPC, PPP.

In the case of Q_{34}, the b-decision is made first, and there are 3 different a-decisions, resulting in 16 different combinations. In total, there are 24 possibilities (in Section 4.4 we assumed that the control variables could only be true or false, but in this case b can have any of 3 values). Those combinations represent different refactorings of the same program. Below, we show a few of them and illustrate the major effect they have on the designs. Even if unlikely combinations such as CCP or CPC are discarded, 8 choices still remain. In a large program, where many conditionals are involved in each one of numerous logical decisions, this can lead to a very large number of choices. In contrast, if MIP is prescribed, there is only one choice.

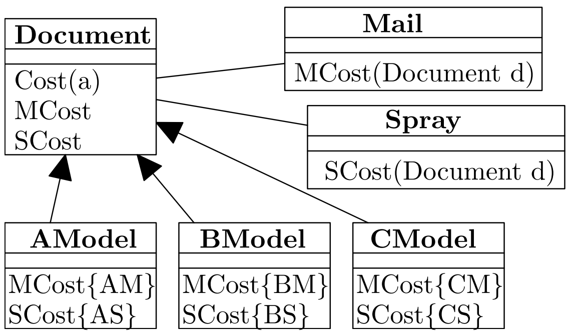


Figure 11: The design corresponding to case CPP of matrix Q_{12} for example E2.

The first design is similar to the running example in the Survey [14]. It corresponds to matrix Q_{12}, case CPP, and is shown in Figure 11. The services are provided by class Document. The caller instantiates one of the Document-derived objects (equivalent to initializing b), and calls its Cost method, passing a. Cost selects between MCost and SCost (the a-decision), then invokes the corresponding polymorphic version (the b-decision), which executes the correct helper code to determine the list of equipment, instantiates a Mail or Spray object, and calls its MCost or SCost method for the final cost calculation, passing itself to make the list available to the method.

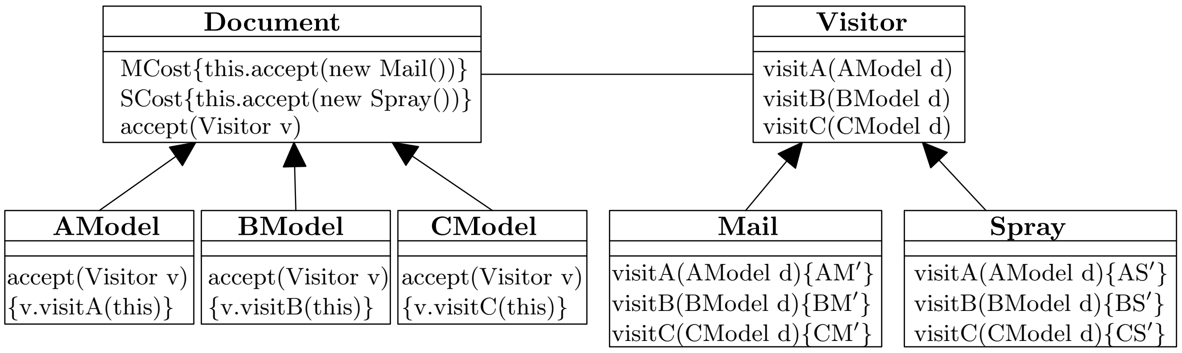


Figure 12: The design corresponding to matrix Q_{34}, case PPPP, of Example E2.

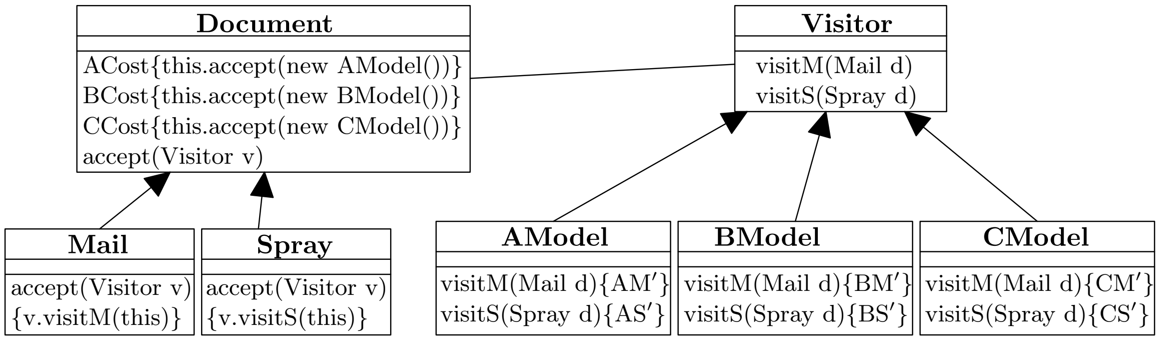


Figure 13: The design corresponding to matrix Q_{12}, case PPP, for Example E2.

The second design is shown in Figure 12. Here, the caller instantiates one of the Document-derived objects (equivalent to initializing b), and calls MCost or SCost, which create either a Mail or a Spray object (equivalent to initializing a). Then, they call the generic accept method (the b-decision), passing the Visitor object just created, and accept invokes the corresponding polymorphic version of visit* (the a-decision). This design corresponds to matrix Q_{34}, case PPPP, because the decisions are made in the order ba and all four are polymorphic. It is important to note that a *Visitor* design pattern [59] was used in the Survey [14] to obtain a similar design, by refactoring the running example, but the transformation took 20 primitive refactorings (14, not counting the 6 renames), all at the level of the OO code, to be compared with one relational operation on matrix Q_{34}.

The last design, shown in Figure 13, is similar to the design of Figure 12, but with the roles of model and task reversed. It is based on matrix Q_{12}, case PPP, because the decisions are made in the order ab and the a-decision and both b-decisions are polymorphic.

All three designs have flaws and are difficult to understand. The design of Figure 11 has mail and spray functionality in all Document subclasses. An upgrade for passenger or cargo transportation would require changes in all the subclasses, plus the addition of a new Passenger or Cargo class, making it even more complex and less understandable. However, upgrading to a new aircraft model D requires the addition of only one subclass. The design of Figure 12 has aircraft model functionality in all Visitor subclasses. An upgrade to model D would require changes in all subclasses, plus the addition of a new subclass to Document. Similar considerations apply to the design of Figure 13, where it is now easy to upgrade to model D, but difficult to upgrade to passenger or cargo.

The use of patterns, as erroneously advocated in the literature [14], does not fix the problems, but merely shifts them to another area. The source of the problems is the destruction of the inherent fundamental symmetry between a and b present in the original matrix Q. When SIP was chosen for implementation, we had to establish an arbitrary order between a and b by using matrices Q_{12} or Q_{23} instead of Q, and symmetry was lost.

6 Conclusions

We have formally presented the Relational Model of Computation (RMC) as a relational database represented by two sparse matrices, where the tuples in the relations are in turn relational models. We have proved the RMC to be a Turing-complete virtual machine, capable of emulating any executable algorithm. We have, therefore, proposed the RMC as a model of computation and as a container for source code. We have pressed the idea that, since the RMC has the characteristics of a matrix, a database, a virtual machine, and a container for code, it shares in the properties of all four. Matrix algebra and relational algebra can be directly applied, and languages such as SQL or Tutorial D [53], or WSL, can be considered. We have presented examples that illustrate the mechanics of the model and some ideas for analyses that can be performed.

Modeling is done by design, not by theory. The modeler must decide which features are important for the problem at hand, and design the model to make them available and easy to handle. A container for source code must be based on a model that is sufficiently flexible and general to allow algorithm designers to represent the features of their choice, to the extent and with the detail they need.

To serve as a container for source code, and in addition to its ability to emulate algorithms engaged in the analysis and transformation of the code, the RMC must be able to model the source code. Being a database, the RMC is ideally positioned for that task. There is no known type of data that can not be represented and handled by a relational database, and techniques for database design are well known. Any information that can be represented in a text document such as source code can certainly be represented in a relational database. Labeled graph representations, such as the ones used for graph rewriting, including CFGs, ASTs and data flow diagrams, discussed in section 4.3, and various other types of representations such as parsed code, token classes, three-address code, single static assignments, and intermediate code representations (IR), used for source code with success, can as well be represented in a database. UML designs and other diagrams can be modelled. Many program entities are directly built into the structure of the RMS. Such is the case for user types, subtypes, single and multiple type inheritance, and several OO entities mentioned in Section 4.5. Hierarchical features extensively used in programs can be described with submodelling.

Traditionally, databases have not been associated with the execution of programs or used to emulate algorithms. The present work demonstrates that databases can do that too. In particular, algorithms that transform or analyze source code can be emulated. The ability of the RMC to represent source code entities with the desired degree of detail, emulate the algorithms that operate on them, and handle large volumes of information, makes the RMC a very attractive container for source code. Emulation of algorithms includes compilers, optimizers, editors and interpreters as well. Using the RMC for compiler optimizations at the same time it is used for source code evolution and analysis, has interesting implications. The RMC becomes the formal repository of the program, and the source code remains as a means for developer-program communication. By separating the communications hub from the formal repository, both can be separately optimized.

In closing, we would like to restate that Equation (1) formally separates the structure of a program, defined by matrices C and Q, from the data of the program, defined by the elements of the two matrices. The concept opens an avenue for research into the design of formal algorithms that use the known structure to operate on the data and transform the program in a mathematically rigorous manner, and communicate with each other through the common RMC interface. The unified approach will promote the development of tools with full capabilities and interoperability, and enhance coherence in program transformation and evolution. These subjects, as well as other areas that currently fall beyond regular computation, are of great interest to us and the possible subjects of our future research.

Biography

Sergio Pissanetzky (born August 9, 1936, Buenos Aires) is an Argentinian physicist mainly active in the United States. He began his studies at Universidad Nacional de La Plata and continued at Universidad Nacional de Cuyo, graduating with a Ph.D. in Physics in 1965. He was a Professor of Physics at Universidad Nacional de Buenos Aires, Universidad Nacional de Córdoba, and Universidad Nacional de Cuyo, and a Research Scientist at Comisión Nacional de la Energía Atómica. Dissatisfied with the political climate of his country, he chose to emigrate. In 1984, in the United States, he joined the Superconducting Supercollider federal project as a Research Scientist, and later, the Graduate School of Physics at Texas A&M University as a Professor of Physics. After a rewarding career as a Scientist, Professor, Entrepreneur, and Consultant, Dr. Pissanetzky retired in 2000.

Dr. Pissanetzky has served as a Member of the Editorial Board of the International Journal for Computation in Electrical and Electronic Engineering, as a Member of the Advisory Committee of the International Journal Métodos Numéricos para Cálculo y Diseño en Ingeniería, and as a member of the International Committee for Nuclear Resonance Spectroscopy, Tokyo, Japan. He has also held positions as a Research Scientist with the Houston Advanced Research Center, as Chairman of the Computer Center of the Atomic Energy Commission, Bariloche, Argentina, and as a Scientific Consultant at Brookhaven National Laboratory and Los Alamos National Laboratory. He was the founder of Magnus Software Corporation, where he focused on development of specialized applications for the Magnetic Resonance Imaging (MRI) and the High Energy Particle Accelerator industries.

Dr. Pissanetzky holds several US and European patents and is the author of three books and many peer reviewed scientific and technical papers. He now lives in a quite suburban neighborhood in Texas, where he spends much of his time doing what he loves: research.

References

[1] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Addison-Wesley Professional, Boston, Massachusetts, 1st edition, October 1998.

[2] E. F. Codd. A relational model of data for large shared data banks. *Comm. ACM*, 13(6):377–387, 1970.

[3] Michael J. Wester (editor), editor. *Computer Algebra Systems - A Practical Guide.* Wiley, Chichester, 1999.

[4] Donald E. Knuth. *The Art of Computer Programming*, volume Vol. 2: Seminumerical Algorithms. Addison-Wesley, 1998.

[5] Christian Bauer, Alexander Frink, and Richard Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *J. Symbolic Computation*, (33):1–12, 2002.

[6] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. *Proc 28th international conference on Software engineering. Shanghai, China*, pages 112–121, 2006.

[7] Harold Thimbleby. User interface design with matrix algebra. *ACM Transactions on Computer-Human Interaction (TOCHI)*, pages 181–236, 2004.

[8] C. Böhm and G. Jacopini. Flow diagrams, Turing machines, and languages with only two formation rules. *Communications of the ACM*, 9(5):266, May 1966.

[9] Zhenchang Xing and Eleni Stroulia. Refactoring practice: How it is and how it should be supported - an Eclipse case study. *22nd IEEE International Conf. on Software Maintenance (ICSM'06)*, 2006.

[10] Curtis Schofield, Brendan Tansey, Zhenchang Xing, and Eleni Stroulia. Digging the development dust for refactorings. *Proc. 14th IEEE Int. Conf. on Program Comprehension (ICPC'06)*, 00:23–34, June 2006.

[11] Suzanne Smith, Sara Stoecklin, and Catharina Serino. An innovative approach to teaching refactoring. *Proc. 37th SIGCSE Technical Symposium on Computer Science Education SIGCSE '06*, 38(1), March 2006.

[12] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1992.

[13] Martin Fowler. *Improving the Design of Existing Code.* Addison-Wesley Professional, Boston, Massachusetts, 1999.

[14] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.

[15] J. Rajesh and D. Janakiram. JIAD: a tool to infer design patterns in refactoring. *Proc 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. Verona, Italy*, pages 227–237, 2004.

[16] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. *Proc. Int. Conf. Software Maintenance*, pages 736–746, 2001.

[17] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, July 2005.

[18] Tom Mens, Gabi Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software Systems Modeling (SoSyM)*, 2006.

[19] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a scripting language for refactoring. *Proc 28th International Conference on Software Engineering. Shanghai, China*, pages 172–181, 2006.

[20] Mark O'Keeffe and Mel Ó Cinnéide. Search-based software maintenance. *Proc. Conf. on Software Maintenance and Reengineering CSMR '06*, pages 249–260, March 2006.

[21] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Engineering*, 20(6):476–493, 1994.

[22] Martin Hitz and Behzad Montazeri. Chidamber and Kemerer's metrics suite: A measurement theory perspective. *IEEE Trans. on Software Engineering*, 22(4):267–271, April 1996.

[23] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. *Proc. European Conf. Software Maintenance and Reeng.*, pages 30–38, 2001.

[24] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. *Proc. Langages et Modèles à Objects*, pages 135–149, August 2002.

[25] Serge Demeyer. Maintainability versus performance: What's the effect of introducing polymorphism? Technical report, Lab. on Reeng., Universiteit Antwerpen, Belgium, 2002.

[26] M. V. Ksenzov. Architectural refactoring of corporate program systems. *Programming and Computing Software.*, 32(1):31–43, January 2006.

[27] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of OO systems. *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering, 2004. (CSMR 2004)*, pages 3–14, March 2004.

[28] Ran Ettinger and Mathieu Verbaere. Untangling: a slice extraction refactoring. *Proc 3rd International Conference on Aspect-oriented Software Development. Lancaster, UK*, pages 93–101, 2004.

[29] Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. *Proc 4th international Conference on Aspect-oriented Software Development. Chicago, Illinois*, pages 135–146, 2005.

[30] Mirko Streckenbach and Gregor Snelting. Refactoring class hierarchies with KABA. *Proc 19th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. Vancouver, BC, Canada*, pages 315–330, 2004.

[31] Marius Marin, Leon Moonen, and Arie van Deursen. An approach to aspect refactoring based on crosscutting concern types. *Proc 2005 Workshop on Modeling and Analysis of Concerns in Software. St. Louis, Missouri*, pages 1–5, 2005.

[32] Alejandra Garrido and Ralph Johnson. Challenges of refactoring C programs. *Proc International Workshop on Principles of Software Evolution. Orlando, Florida*, pages 6–14, 2002.

[33] Michel Dagenais, Ettore Merlo, Bruno Lagu, and Daniel Proulx. Clones occurence in large object oriented software packages. *Proc. 1998 Conf. of the Centre for Advanced Studies on Collaborative Research. Toronto, Ontario, Canada*, page 10, November 1998.

[34] Salah Bouktif, Giuliano Antoniol, Ettore Merlo, and Markus Neteler. A novel approach to optimize clone refactoring activity. *Proc.8th Annual Conf. on Genetic and Evolutionary Computation. Seattle, Washington, USA*, pages 1885 – 1892, July 2006.

[35] Hamid Abdul Basit, Damith C. Rajapakse, and Stan Jarzabek. Beyond templates: a study of clones in the STL and some general implications. *Proc. 27th Int. Conf. on Software Engineering. St. Louis, MO, USA*, pages 451 – 459, May 2005.

[36] Ira D. Baxter, Andrew Yahin, Leonardo de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. *Proc. Int. Conf. on Software Maintenance, ICSM'98*, pages 368–377, March 1998.

[37] James R. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation (keynote). *Proc. 11th IEEE Int. Workshop on Program Comprehension (IWPC'03)*, page 196, 2003.

[38] M. Balazinska, E. Merlo, N. Dagenais, B. Laguë, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. *Proc. Working Conf. Reverse Eng.*, pages 98–107, 2000.

[39] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. ARIES: Refactoring support tool for code clone. *Proc. Third Workshop on Software Quality. St. Louis, Missouri, USA*, pages 1 – 4, May 2005.

[40] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[41] Jens Krinke. Identifying similar code with program dependence graphs. *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, pages 301–309, October 2001.

[42] Stephane Ducasse, Oscar Nierstrasz, and Matthias Rieger. Lightweight detection of duplicated code. A language-independent approach. *IAM-04-02*, pages 1–30, February 2004.

[43] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. *Automated Software Eng.*, 8:89–120, 2001.

[44] Jean-Marie Favre. Preprocessors from an abstract point of view. *Proc. International Conf. on Software Maintenance*, pages 329–338, November 1996.

[45] Diomidis Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Trans. Software Eng.*, 29(11):1019–1030, November 2003.

[46] B. McCloskey and E. Brewer. ASTEC: a new approach to refactoring C. *Proc. 10th. European Software Engng. Conf., Lisbon, Portugal*, pages 21–30, 2005.

[47] M. P. Ward and K. H. Bennett. Formal methods to aid the evolution of software. *Int. J. Software Eng. and Knowledge Eng.*, 5(1):25–47, 1995.

[48] James E. Smith and Ravi Nair. *Virtual Machines. Versatile Platforms for Systems and Processes.* Morgan Kaufmann. Amsterdam, 2005.

[49] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, September 1992.

[50] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.

[51] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation, Vol 148, pp. 1-70, 1999*, 148:1–70, 1999.

[52] Sergio Pissanetzky. *Sparse Matrix Technology.* Academic Press, London, 1984.

[53] C. J. Date and H. Darwen. *Databases, Types, and the Relational Model. The Third Manifesto.* Addison-Wesley, Reading, Massachusetts, third edition, 2006.

[54] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, Boston, Massachusetts, 1997.

[55] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[56] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices, Proceedings of a symposium on Compiler optimization, Urbana-Champaign, Illinois, 1970*, 5(7):1–19, 1970.

[57] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, March 2004*, 2004.

[58] Sergio Pissanetzky. *Refactoring with Relations. A new Method for Refactoring Object-Oriented Software.* SciControls.com, Texas, USA, July 2006.

[59] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems.* Addison Wesley, 1994.