DM

Organizations Redesign and Building of Information Systems

MASTER DISSERTATION

Duarte Paulo Brazão Gouveia

MASTER IN INFORMATICS ENGINEERING

A Nossa Universidade

www.uma.pt

December | 2014

Organizations Redesign and Building of Information Systems

MASTER DISSERTATION

Duarte Paulo Brazão Gouveia
MASTER IN INFORMATICS ENGINEERING

SUPERVISOR
David Sardinha Andrade de Aveiro

Producing **A**pplications with **X**HTML

Organizations **R**edesign and **B**uilding of **I**nformation **S**ystems

Templated **E**xtendable **R**esources for **R**apid **A**pplication **R**euse, **U**pdate and **M**aintenance

This work aims at building web applications as Information Systems for organizations in a process such that they can be produced in a easy and rapid way and have the ability to be changed as the organization evolves over time.

These web applications use templates of resources that can be refined, adapted and extended over time to facilitate adaptation and maintenance.

PAX ORBIS TERRARUM

PAX ORBIS TERRARUM

Figure 1 - Otho Denarios - roman coin from the emperor Otho

The latin expression **Pax Orbis Terrarum** means **universal peace** or, in literal terms, peace (pax) in all the land (terrarum) that surrounds us (orbis). This expression was used in coins of the Roman Empire. The silver coin in the picture is an Otho Denarius, from the emperor Otho. Marcus Otho Caesar Augustus was the 7th emperor of the Roman Empire in the year 69 AD and committed suicide to prevent a civil war in the Roman Empire after only 3 month as emperor. Otho was governor of the Province of Lusitania for 10 years just before becoming emperor. Lusitania was the Roman Province where today Portugal is.

Otho immediately followed the 6th emperor Galba Caesar Augustus from Tarraconensis in Spain, that was emperor for 7 months. Galba once said that in the west of the Iberian Peninsula lived a people that was unable to rule themselves but didn’t allow others to rule them.

The results of this work is an architecture for Information Systems in organizations that lets them feel in peace and in control of their domain.

POT

POT

Data POT

Action POT

Figure 2 - Clay Pots from ancient times

The building blocks of this architecture are Data Pots and Action Pots. Like the old clay pots of ancient times they are multi-use containers that preserve content and facilitate its content use when appropriate.

Resumo

Desenvolver software continua a ser uma atividade económica de risco. Mesmo com 60 anos de experiência, a nossa comunidade continua a não ser capaz de construir sistemas de informação para organizações de forma consistente, com qualidade previsível, dentro dos orçamentos e dos calendários acordados.

O principal objetivo deste trabalho é criar uma nova forma de desenvolver sistemas de informação para organizações, com tecnologia internet, de uma forma mais rápida, melhor, mais barata e que seja mais adequada para lidar com a mudança organizacional.

Para o fazer propomos um método de crescente formalização com oito degraus. Para cada área de atividade as organizações podem adotar um nível de formalismo e ao longo do tempo evoluir em cada área do sistema de informação, tornando-o mais ou menos formal consoante a evolução do negócio.

Propomos uma arquitetura global, prevendo que utilizador fará uso de múltiplas interfaces em simultâneo, para lidar com várias organizações numa mesma interface lógica. A arquitetura inclui um agente agregador da informação por utilizador a quem se pode delegar atividades de coordenação. No núcleo da arquitetura estão as aplicações com os seus dados e ações, alguns privados outros partilhados.

Propomos um modelo de objetos adaptativo que utilize uma nova ontologia para dados e ações com regras estritas de normalização. Estas regras restringem os efeitos das mudanças tornando-as mais testáveis e mais facilmente corrigíveis e adaptáveis.

Propomos uma evolução no padrão universal de transações da Engenharia Organizacional, que acrescenta aos atos de coordenação e produção, os atos de conhecimento e de significado. Propomos ainda uma transformação nos atos de coordenação, tornando-os mais flexíveis e adaptados aos processos negociais que visam o alcançar de acordos entre as partes.

Este trabalho implementa um protótipo com parte do sistema proposto de forma a ter uma avaliação preliminar da sua possibilidade de construção e limitações.

Palavras chave: Sistemas de Informação; Engenharia Organizacional; Mudança Organizacional; Modelos de objetos adaptativos; Ontologias Organizacionais; Sistemas Normalizados

Abstract

To develop software is still a risky business. After 60 years of experience, these community is still not able to consistently build information systems (IS) for organizations with predictable quality, within previously agreed budget and time constraints.

The main goal of this work is to create a new way to develop flexible IS for organizations, using web technologies, in a faster, better and cheaper way that is more suited to handle organizational change.

To do so we propose a formalization ladder with eight steps. Each organizational area can adopt a specific formalization level and, over time, evolve its information system making it more formal or less formal according to the business current needs.

We propose a global architecture, allowing the user to use several interfaces simultaneously to deal with IS of several organizations in the same logic interface. The architecture includes an agent per user that aggregates the information and performs coordination acts by delegation. At the core of the architecture are the applications with their data and actions, some private, others shared.

We propose an adaptive object model that uses a new ontology for data and action with strict normalizing rules. These rules bound the effects of changes so that they can be better tested, corrected and evolved.

We propose an evolution to the universal pattern for transactions from Organizational Engineering that adds knowledge and meaning acts to the already existing production and coordination acts. We propose a transformation to the coordination acts, making them more flexible and adapted to negotiation processes in which parts aim to reach an agreement.

This work implements a prototype with part of the proposed system in order to have a preliminary assessment of its feasibility and limitations.

Keywords: Information Systems; Organizational Engineering; Organization Change; Adaptive Object Model; Enterprise Ontology; Normalized Systems

List of Figures

Figure 1 - Otho Denarios - roman coin from the emperor Otho.. ii
Figure 2 - Clay Pots from ancient times iii
Figure 3 – Mesh of related topics...xi
Figure 4 - Roman Empire at maximum extent in 150 CE 2
Figure 5 - The cycle in “The Structure of Scientific Revolutions” (1962) by Thomas Kuhn [12] ... 4
Figure 6 – Foundational Theories for Organizational Engineering [4] 17
Figure 7 - Arity of operations in Universal Algebra 18
Figure 8 - Venn diagram illustrating some set operations 19
Figure 9 - Elements of a graph 20
Figure 10 - Examples of Assiciative Maps 23
Figure 11 - B-Tree [63].. 24
Figure 12 - Elements of a System 24
Figure 13 – PSI theory - Social Interaction Pattern [65] 25
Figure 14 - 3 fundamental questions to determine the nature of relations, based on [66]....... 27
Figure 15 - Neuron Model 28
Figure 16 - TOPO global architecture 33
Figure 17 - Communication state chart... 40
Figure 18 - Structural elements of Zachmann Framework 43
Figure 19 - Usual sets of numbers in mathematics 51
Figure 20 - Indexing structure for infinite size identifiers 56
Figure 21 - Construction to hold Universal Data Structures 61
Figure 22 and 23 – Instantiation of simple list and ring structures 62
Figure 24 - Binary tree representation... 63
Figure 25 – Instantiation of a binary tree structure... 63
Figure 26 - Adapted MOESI shared memory state transition diagram (Used in AMD Opteron) 64
Figure 27 - Visual representation of a Turing Machine.. 72
Figure 28 – Prototype for message handling and association of Zachmann Dimensions........... 76
Figure 29 - Distribution of Dots according to prime numbers 77
Figure 30 - Prototype for the handling of concepts (Person) 78
Figure 31 - Prototype for the visualization of complex structures (Clusters of Persons) 79
Figure 32 - Prototype for visualization of transactions using DEMO 80

List of Acronyms

- API – Application Programming Interface
- CPU – Central Processing Unit
- DEMO – Design and Engineering Methodology for Organizations
- IS – Information Systems
- LOC – Lines Of Code
- MMX - Multiple Math eXtension
- MOF – Meta Object Facility
- NASA – National Aeronautics and Space Administration
- PHP – PHP Hypertext Preprocessor
- POSET – Partially Ordered Set
- PSI Theory – Performance in Social Interactions Theory
- SATA – Serial ATA International Organization
- SIMD – Single Instruction Multiple Data
- SMS – Short Message Service
- TOPO - Transparent Open Platform for Ontologies
- UML – Unified Modeling Language
- XHTML – Extensible Hyper Text Markup Language

Index

1. Introduction 1
  1.1 Organizations 1
  1.2 State of Information Systems field 2
    1.2.1 An analogy with the Roman Empire 2
    1.2.2 Fragmentation and crisis 3
    1.2.3 A new paradigm – Organizational Engineering 3
  1.3 The structure of this thesis 5
2. Motivation, Goals and Axioms 7
  2.1 Motivation 7
  2.2 Goals 7
  2.3 Axioms 8
3. Problem Statement 10
  3.1 Change in IS for organizations 10
  3.2 Other problems in IS 12
    3.2.1 Complexity 12
    3.2.2 Conformity 13
    3.2.3 Invisibility 14
    3.2.4 Performance 14
    3.2.5 Correction 16
  3.3 Existing approaches to handle change in IS 16
    3.3.1 Agile 16
    3.3.2 Frameworks 16
4. Review of Literature 17
  4.1 Theoretical Framing 17
  4.2 Philosophical Theories 18
    4.2.1 Universal Algebra 18
    4.2.2 Set Theory 19
    4.2.3 Map Theory 20
    4.2.4 Graph Theory 20
    4.2.5 State Machines 21
    4.2.6 State Charts 21
    4.2.7 Data and Metadata 22

4.2.8 Basic Data Types ... 22
4.2.9 Common Complex Data Structures ... 23
4.2.10 Automata ... 24
4.2.11 Systems ... 24
4.3 Ontological Theories ... 25
4.3.1 Performance in Social Interactions Theory ... 25
4.3.2 Foundations Ontologies ... 26
4.3.3 Relations Nature ... 26
4.3.4 Neuron Model ... 28
4.4 Ideological Theories ... 28
4.4.1 Business Motivation Model ... 28
4.4.2 Values ... 28
4.5 Technological Theories ... 29
4.5.1 Normalized Systems ... 29
4.5.2 Functions ... 29
5. Options ... 31
5.1 Scope and Technology ... 31
5.2 Performance ... 31
5.3 Correction and Robustness ... 31
5.4 Control, Security and Reliability ... 32
5.5 Usability ... 32
6. Vision ... 33
6.1 TOPO Architecture ... 33
6.1.1 Persons ... 34
6.1.2 Interfaces ... 35
6.1.3 Agents ... 36
6.1.4 Applications ... 37
6.1.5 Data Cells ... 37
6.2 Asynchronous Message Model ... 38
6.3 Memory System ... 38
6.4 Transaction Protocol ... 38
6.5 Neuron model with Zachman Framework Dimensions ... 42
6.6 Future Programming ... 43
7. Existing and Proposed Solutions ... 48

7.1 Bottom-Up approach 48
7.1.1 Basic Data Types 48
7.1.2 Concepts, Objects and Types 54
7.1.3 Identifiers 55
7.1.4 Basic Data Pot 57
7.1.5 Structured Data Pot 60
7.1.6 Universal Data Structure 60
7.1.7 Shared Data Pots 64
7.1.8 Visual Programming 65
7.1.9 Index Data Pot 65
7.1.10 Basic Action Pot 66
7.1.11 Mapping Action Pot 71
7.1.12 Strategy Action Pot 71
7.1.13 Visual Action Pot 72
7.1.14 Grammar Action Pot 72
7.1.15 Turing Machine Pot 72
7.2 Method: Formalization Ladder 73
7.2.1 Step 1 - No Formalization (big ball of mud) 73
7.2.2 Step 2 - Adding Dimensions with neurons 74
7.2.3 Step 3 – Adding Attributes 74
7.2.4 Step 4 – Adding Concepts 74
7.2.5 Step 5 – Adding Constraints 74
7.2.6 Step 6 – Adding Structural Relationships between concepts 75
7.2.7 Step 7 – Adding Transactions 75
7.2.8 Step 8 – Delegating tasks 75
7.3 Analysis of Ladder Formalization 75
7.4 Prototype - Interface 76
7.5 Prototype – Inner Structure of Data 80
8. Functionalities and Requirements Analysis 85
9. Analysis using Design Science Research Guidelines 89
Guideline 1: Design as an Artifact 89
Guideline 2: Problem Relevance 89
Guideline 3: Design Evaluation 89
Guideline 4: Research Contributions 89

Guideline 5: Research Rigor 89
Guideline 6: Design as a Search Process 90
Guideline 7: Communication of Research 90
10. Conclusions 91
11. Time and Other Constraints 93
12. Ongoing and future work 93
13. References 94

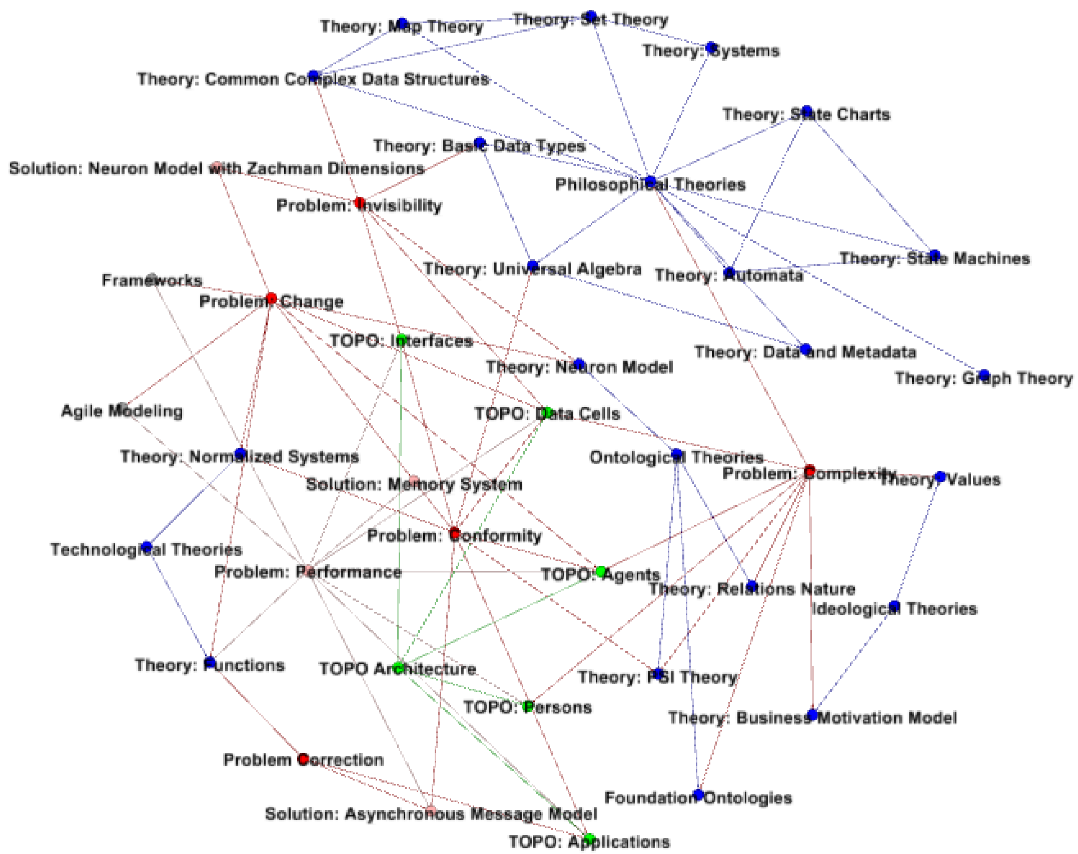


Figure 3 – Mesh of related topics

1. Introduction

The title of this work has an embedded game of words.

Producing **A**pplications with **X**HTML **O**rganizations **R**edesign and **B**uilding of **I**nformation **S**ystems **T**emplated **E**xtendable **R**esources for **R**apid **A**pplication **R**euse, **U**pdate and **M**aintenance

On a first glimpse it's just a long sequence of technical terms that says "***what***" this work is all about – an engineering layer that addresses the building of information systems (IS) for organizations.

With the first letter of each word, we get the Latin expression "Pax Orbis Terrarum" that leads to the "***why***" question through the analogy presented below – a meaning layer with the deeper purpose of this work.

PAX **O**RBIS **T**ERRARUM

Repeating the first letter simplification again we get a single word "POT" of a real world artifact and again an analogy to "***how***" we will try to solve the main problem addressed – the need to handle change in IS for organizations.

POT

This introduction section addresses the following topics:

1) The subject of study: organizations
2) The state of IS field of study: analogy; fragmentation and crisis; new paradigm
3) The structure of this thesis: IMPROVE FACTOR

1.1 Organizations

Organizations are social structures whose members communicate and coordinate their behavior in order to accomplish shared goals or to put out valuable goods (ideas, environmental conditions, products or services). Organizations can produce goods for external entities or for their own consumption – becoming "prosumers" [1].

A few years ago, the concept of organizations was restricted to relatively big, rational, formal and structured entities with a predefined set of people, fixed roles, procedures and rules like schools, hospitals, factories, governments and businesses. Richard Scott extended this concept to a much broader sense, including what he called **natural systems** and **open systems** [2][3].

Natural systems are organizations "whose participants pursue multiple interests, forged in conflict and consensus, but who recognize the value of perpetuating the organization as an important resource" [3].

In open systems, "organizations are congeries of interdependent flows and activities linking shifting coalitions of participants embedded in wider material-resource and institutional environments" [3]. Structures like families, social groups, movements, friendship cliques, and other dynamic structures can also be seen as organizations – even a single person can be seen as an organization.

Nowadays, people live surrounded by organizations and networks of organizations that interact in a complex environment. Organizations are organized complexities [4] that face a major challenge – the need to handle the continuous change in their environment.

1.2 *State of Information Systems field*

1.2.1 An analogy with the Roman Empire

The Latin sentence “Pax Orbis Terrarum”, also derived from the title of this work, is an expression that means “universal peace” or literally, peace (pax) for all the land (terrarum) that surrounds us (orbis).

The reason for this reference to the Roman Empire is a useful analogy between what happened in the lands around the Mediterranean (“mare nostrum” – “our sea”) before and during the Roman Empire, and the current state of the information systems field.

Before the Roman Empire, people lived in small communities in subsistence economy with little commerce, inefficient and small markets, and minute interaction with other communities besides nearby neighbors. Each community had its own ruler and lived according to their own rules. Roads between communities were bad or inexistent. War was the recurring state, as pillage was an easier way to get better lands and goods.

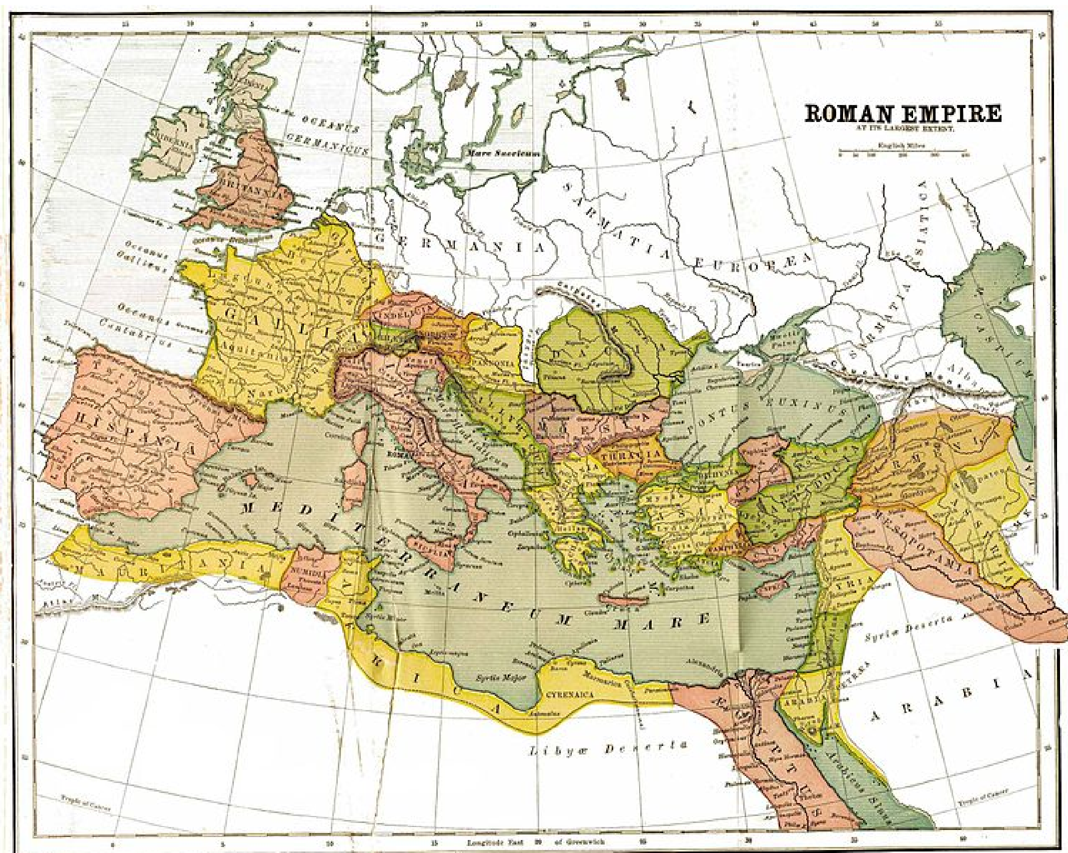


Figure 4 - Roman Empire at maximum extent in 150 CE

The Roman Empire lasted 5 centuries. The first 2 centuries were called “Pax Romana” (“Roman peace”) due to the relative stability that the Roman Empire was able to bring without precedent until that time.

Although we have a clear image from the roman armies built around copper, coal, leather and organization, it was the Lex Romana across the empire (“Roman rule of law”) based on the Latin language, and the ability to enforce it with the legions, that brought the stability to that empire.

The city of Rome itself, 20 centuries ago, had around 1 million people. More than double the second largest city at that time, and that record was only surpassed in the XIX century by London. This was only possible by an efficient economy based on commerce in wide

organizations based on efficient networks of trade. It is reported that the transportation of grain from Hispania (“Spain”) to Rome would only increase 16% to the product cost [5].

Roman engineering artifacts still stand as a proof of their advances with aqueducts, mills, dams, bridges, roads, theaters, coliseums, pantheons, baths, mines and water systems.

1.2.2 Fragmentation and crisis

As an analogy, the current state in the field of Information Systems is the world before the existence of the Roman Empire. According to Klein [6], the IS field suffers from two problematic structural patterns:

- A state of fragmentation
- Significant communication gaps

The author argues that fragmentation is different from diversity, as the last one presumes the existence of a common base that currently does not exist in IS field.

Avison [7] argues about the lack of discipline in IS field as there is no agreed general area for teaching, research and practice. This leads to a perceived lack of coherence in the discipline and a low status as a consequence. The theoretical foundations of IS come from a diversity of theories from many “reference disciplines, including economics, mathematics, linguistics, semiotics, ethics, political science, psychology, sociology and statistics”. These multiple theories may be “mutually inconsistent” and generate many unconnected “schools of thought with their own metatheoretic assumptions, research methodologies and adherents”.

Benbasat [8] claims that in order to have identity in the IS field there must be: central character; claimed distinctiveness and temporal continuity. Galliers [9] points toward the need of a trans-disciplinary view of IS in order to set boundaries, define the scope, focus on the central artifact and its properties.

Klein [6] argues that a common language is needed for overcoming the communication gaps, and based on that solid ground develop a common body of knowledge for IS.

Orlikowski [10] regrets the “blind spot” in the IS field of ignoring, or taking for granted, the conceptualization or theorizing of the artifact. On the other hand, Lyytinen [11] argues that not having a core theory at the center of the IS field is not a reason for concern, as long as relevant results are produced. We share the view that current results are not solid enough to argue in favor of no reason to concern…

The visions about the path to be followed are not fully consistent, as you would expect of a fragmented world. But in one thing they all agree: the IS field is in crisis and actions should be taken in order to get to a better ground. But how can we do that?

1.2.3 A new paradigm – Organizational Engineering

Kuhn in “The Structure of Scientific Revolutions” [12] presents a cyclic model for how science works.

According to Kuhn, “normal science” follows Karl Poper’s “moralistic methodology” [12] toward truth that states that “science proceeds by conjectures and refutations”.

This method consists of:

1. Frame bold conjectures that can be falsifiable, as testable as possible.
2. If they are refuted, find new conjectures that fit the observable phenomena’s.

Kuhn proposes a different model:

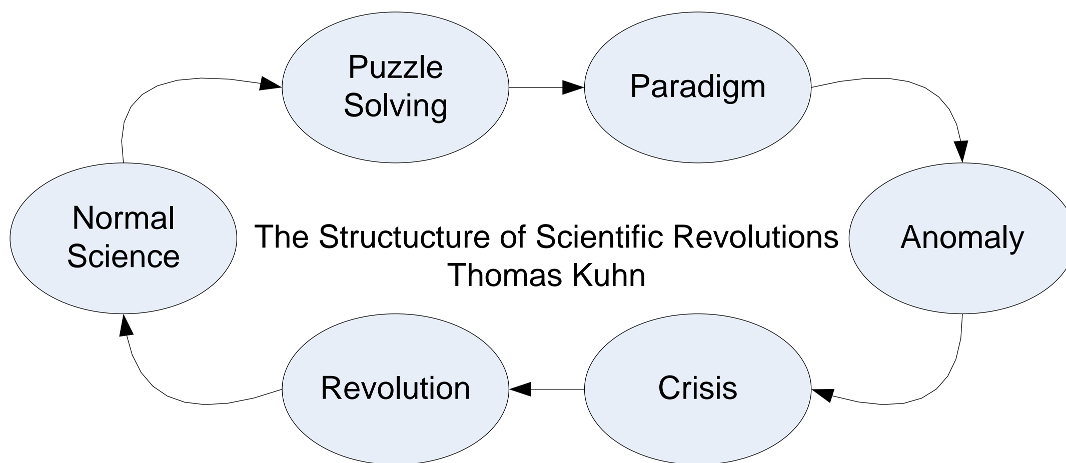


Figure 5 - The cycle in "The Structure of Scientific Revolutions" (1962) by Thomas Kuhn [12]

Kuhn though that the usual way science progresses is just like solving puzzles that are left open within the current field of knowledge – that is, within the current set of theories that establish a paradigm and dominant strategies to solve those puzzles he called "normal science". "The most striking feature of normal research problems we have just encountered is how little they aim to produce major novelties, conceptual or phenomenal." According to Kuhn, normal science addresses 3 types of problems:

- **Determination of significant facts**

  Measure phenomena to better describe and predict according to current theories.

- **Matching of facts with current theory**

  Adjust current theory to match the observed facts when discrepancies occur.

- **Articulation of theory**

  Elaborate on the implicit consequences of current theories and applications.

There can co-exist many competing theories in the same field event when huge anomalies are detected due to mismatch between existing theories and observed phenomena. That is certainly the current situation in the construction and maintenance of IS for organizations, as stated above.

Kuhn says that scientists won't leave their current theories until a new, sufficiently better paradigm is able to replace it. That is the revolution moment: "The decision to reject one paradigm is always simultaneously the decision to accept another, and the judgment leading to that decision involves the comparison of both paradigms with nature and with each other."

Organizational Engineering hopes to be a paradigm shift with a new, holistic and systemic approach to address change in organization of all sizes [4]. In this work we use the term "organizations" instead of the also common term "enterprise", as we believe that the principles hereby described have broader appliance than just to organizations oriented to business.

These principles for organizational engineering have been established in a manifesto [13], that, in my perspective, aims to the following goals:

1. **Holistic view of organizations**

   Deliberate design, engineering and implementation of organizations as unified and integrated wholes.

2. **Organizations as social systems**

   People are at the core of organizations being their owners, users, workers and beneficiaries. People coordinate their acts through messages and establish commitments in universal patterns called transactions.

3. **Black-box and white-box perspectives**

   Black-box perspective is the functional (or teleological) perspective that each user establishes with artifacts in use (which can be different from what they were designed for).

   White-box perspective is the construction (or ontological) perspective that artifacts have based on their objective reality.

4. **Information system ontology**

   Information system ontology is the meta model for designing organizations using atomic elements in its construction. This ontological model should come prior to any functional model that describes its usage.

5. **People with authority and responsibility**

   People, as intelligent beings, should take control of their organizations as they get authorized to perform roles and assume responsibility for their actions. They are in better position to act in the best interest of the organization than the initial conceptual designers of the organizations IS that are not able to embrace the full complexity of everyday environment that organizations operate in.

6. **Organizations architecture**

   Organizations should be driven by strategic concerns and values that in time are transformed into normative principles – the culture and the way things work in each organization.

7. **Organizations governance**

   In order to manage change as an integrated unity, organizations have to establish and apply measures of control.

1.3 The structure of this thesis

This thesis is structured into 13 sections following the acronym: **IMPROVE FACTOR**. We hereby describe the content and contribution that each section brings to this work.

1. **I**ntroduction

   The current section introduces: the subject of study: organizations; analysis of current state of the field; and the structure of this thesis.

2. **M**otivation, Goals and Axioms

   Explains the motivation for this work and its main goals, leading to a research question and a list the axioms – the assumptions being made as starting points for reasoning.

3. **P**roblem Statement

   Deepens the analysis of the problems posed by change in IS for organizations - the main problem addressed with this work. Defines the scope by stating what will be handled and other challenging problems in the subject of study not addressed by this work. Identifis the existing approaches to handle the change in building IS.

4. **R**eview of Literature

   Succinct description of the support theories used for this work, analyzing separately the philosophical, ontological, ideological and technological theories.

5. **O**ptions

   Defines the options for this architecture, by defining the non-functional requirements in terms of scope, technology, performance, correction, robustness. control, security, reliability and usability.

6. **V**ision

   Top-down approach for the proposed solution, including all the core structural decisions like the global system architecture, called TOPO with its components (persons, interfaces, agents, applications and data cells); the asynchronous message model; the memory system; the transaction protocol; the neuron model with Zachman framework dimensions; future visual programming.

7. **E**xisting and Proposed Solutions

   Bottom-up approach with comparison between existing and proposed solutions to implement the architecture, from the most basic data types to the most complex data and action structures. Also an eight-step method to allow organizations to gradually move up the formalization ladder in order to improve their information system and adapt it to their needs.

8. **F**unctionalities and requirements analysis

   Analytical evaluation of how each requirement proposed in section 5 is addressed by the solutions proposed in sections 6 and 7, and its level of success in complying with those requirements.

9. **A**nalysis using Design Science Research Guidelines

   Analysis of current work using Design Science Research Guidelines.

10. **C**onclusions

    The lessons learned from this work.

11. **T**ime constraints

    This work limitations.

12. **O**ngoing and future work

    Possible alternatives as future work.

13. **R**eferences

    List of references in this thesis.

2. Motivation, Goals and Axioms

This section contribution:

> Explain the motivation for this work and its main goals, leading to a research question and a list the axioms – the assumptions being made as starting points for reasoning.

2.1 Motivation

This work motivations is the belief that it is possible to create a information system ontology, as defined in [14], that is generic enough to be used to model the world, but at the same time concrete enough to enable its automatic transformation into working prototypes, or even full software applications. This belief is stemmed from the author's professional experience in developing code generation tools that produce fully working database applications from simple entity-relationship models.

This work aims to create a paradigm shift [12] on software development by taking the initial steps into the creation of a theory, a information system ontology and a method for developing IS for organizations, one order of magnitude better in the aggregate of all improvement factors defined in chapter 5. This is an imprecise measurement, because paradigm shifts are usually subject to incommensurability [12] – that is, the impossibility of accurately compare with solutions in the previous paradigm. Our hope is that the benefits are so clear that the fulfillment of these goals become self evident to the users and stakeholders.

Aiming one order of magnitude improvement means that we are aiming to create a "Silver Bullet", as defined in 1956 by Frederick Brooks in the classic paper "No Silver Bullet: Essence and Accidents of Software Engineering" [15], with the interpretation given by the same author in the preface of [16].

2.2 Goals

The ultimate goal of building software for organizations is to help users to be better by deciding, producing, consuming and remembering better. Being humans we have flaws in our judgments. Science has studied many cases of our biased decisions [17][18][19].

Most of our daily decisions are based on simple rules that are really just simple models. Probably the most common model of them all is to copy what others are choosing as decision criteria – there is safety in big numbers. But in the future, people will rely more on software tools to better model the world, be able to make informed and rational decisions (unbiased) and, in time, become better than our nature.

Building software, big or small, is about creating models of reality. Every model is a simplification, so a lot is lost in the modeling process. Accumulating knowledge is not the purpose of an IS. The real purpose is to make things more manageable by making things simpler through conceptualization.

A software model is useful for analyzing the current state of the world by querying or viewing synthetic reports, for predicting the future or retrodicting – predicting the past and learn from experience.

Models represent world that can be continuous or discrete. Actions performed can be deterministic or stochastic, that is, unpredictable – especially if the model does not take account with all possible actions that are happening in the world. Information about the world might not even be available (or with required precision). The roles of the several persons in the world can differ and change over time from collaborative mode to adversarial or even irrelevant interaction.

This inconsistent and dynamic world may lead to the idea that the de-facto standard pattern of software architecture, "big balls of mud" [20], might be adequate, especially for organizations

that are mostly natural systems or open systems, as defined by Richard Scott [2]. As Niccolo Maquiavel (1949-1527) puts it: “Nothing is more difficult than to change the order of things.”

This work aims to mitigate the lack of flexibility of software applications felt by persons and organizations when faced with unforeseen situations or change. Therefore, the research question for this work is: can an IS artifact be built satisfying these goals?

2.3 Axioms

Axioms are the premises or starting points for reasoning. Axioms are supposed to be self evident that can be accepted as true without controversy, although that can be a hard assumption in a fragmented field of study like IS.

Theories are the best explanations to a certain phenomena. Theories are built over sets of axioms. This work assumes as true the following axioms:

Axiom 1. People are the core element of any information system.

The core element is not the data or the events but the people that make sense of that information. Information is teleological usage that people get from world facts. Information is “the difference that makes a difference” [3] to people. Each person can have a level of authority and responsibility in the information system allowing them to take responsible decisions.

Axiom 2. Change is inevitable.

Long gone are the times of Parmenides (515BCE – 460BCE) that argued that change is not possible. Since Heraclites (535BCE – 475BCE) that change is assumed to be ever-present as stated “No man ever steps in the same river twice”. Not only the water in the river is not the same, but also the person is not exactly the same as biology changed and the person gained experience from the previous steps.

Axiom 3. The world is more complex that what can be modeled by any IS.

António Damásio has argued [21] that the reason why our brains are so complex is because we need to handle ambiguity. Simply combining the elements in several dimensions (who, what, when, where, how) is a model simple enough to map the brain with its neuron like structures, but without the enough power to model it’s use.

Axiom 4. The world is continuous. Perception of the world is discrete.

There are two types of changes [22]: the continuous change in the world and the change in the perception that is discontinuous and always received as an inconsistency with the previous mental model of the world.

Axiom 5. The world is stochastic (non deterministic).

No person action can determine the future state of the world because there are many other uncontrolled actions in the world. The information system model may be deterministic, leading to inconsistencies between the world and its model.

Axiom 6. Persons can collaborate, oppose or be neutral to your goals.

Open organizations, as defined by [2], are shifting coalitions of participants that change their goals and coordination strategies over time.

Axiom 7. Communication between persons is enough to manage any IS

People were able to build up complex organizations for centuries. Even before writing, the only mean they had was interpersonal communication. These means that communication between agents with memory alone is enough to manage any IS.

Axiom 8. Communication is asynchronous.

Even in face to face communication the messages are sent and perceived asynchronously. Synchronizations is a simplification that is not supported by the real world.

Axiom 9. IS for organizations must be holistic and live long lives.

An IS must handle the broad environment the organization lives in and must be able to grow and shrink as the needs of the organizations change over their lifecycle.

3. Problem Statement

This section contribution:

> Deepen the analysis of the problems posed by change in IS for organizations - the main problem addressed with this work. Define the scope by stating what will be handled and other challenging problems in the subject of study not addressed by this work. Identify the existing approaches to handle the change in building IS.

3.1 Change in IS for organizations

Change creates major difficulties to organizations in order to cope with the renewed operational environment. According to Dietz [4], the success perspectives of enterprise strategic initiatives are not favorable because organizations usually do not achieve the expected results. Several sources [24][25] show rates of success lower than 10% or 30%, arguably because of gaps between those initiatives and the way people and organizations learn and take decisions.

Organizations face increasing challenges in data demand and data supply. In terms of data demand, the increase of traffic had an inflection point in 2007 with the launch of iPhone and the start of a new phase in data access from mobile devices with demand for traffic doubling every year [26].

In terms of data supply, today's organizations can generate big amounts of internal data in very little time – much faster than what can be understand without the proper tools. Companies like Google and Facebook have reported a substantial increase on the amount of data being produced every year, but those number are hard to compare to the pre-computer age as they are built from so different paradigms [12].

On the storage side Kryder's law [27] state that "magnetic disk areal storage density is increasing very quickly at a pace much faster than doubling in every 18 months in Moore's law". Although disk capacities are growing fast, disk latencies and data transfer rate are not keeping the pace. Currently, a computer can only get 300Mb/s from one disk with 3.0 SATA.

Organizations need Information Systems (IS) that can enable Data Science, that is, [3] "the ability to take data – to be able to understand it, process it, to extract value from it, to visualize it, to communicate it". Organizations need tools that allow them to manage their business models, their data and metadata and cope with their changing environment.

In order to cope with increased organization complexities the amount of lines of code (LOC) in software is increasing at an impressive rate [28] – doubling every ten months [29] – which is almost double of Moore's law. So many millions LOC will inevitable have errors because there is still no economic way to perform formal verifications automatically and testing. As Dijkstra [30] said it, tests "shows the presence, not the absence of bugs". The biggest effort of formal verification had less than 10 000 LOC for a operating system kernel and its cost was $500/LOC[31][32]. As a reference point, software for NASA typically costs $80/LOC.

Lehman's eight laws of software evolution [33] emphasize the inevitability of change in IS and their effects:

- ***Law of continuing change (I)***

  "A software that is used must be continually adapted else it becomes progressively less satisfactory."

- ***Law of increasing complexity (II)***

  "As a program is evolved its complexity increases unless work is done to maintain or reduce it."

- ***Law of continuing growth (VI)***

  "Functional content of a program must be continually increased to maintain user satisfaction over its lifetime."

- ***Law of declining quality (VII)***

  "Software will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment."

- ***Law of feedback system (VIII)***

  "Programming processes constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved"

In order to check the impact of change we looked at change logs for PHP version 4 and 5 that are freely available online [34][35]. PHP is one of the most popular programming languages with constant support and monthly updates.

For the 9 years lifetime of PHP4 (Jul99-Aug08) there were 2484 bugs fixed/improvements made with an average of 23 items per month. For the current version PHP5, that is already 10 years old (Jun03 - Sep2013), there have been 5648 bugs fixed/improvements made, with an average of 46 items per month. We might think that the number of bugs diminish as time passes by, but that is not the case for this case where the numbers remain stable. In the last 3 years there where on average 47 items changed in new releases per month.

On the day of this analysis (3Sep2013), if we look at stats [36] we will find out that there were 65257 issues reported. Only 4% of the reported issues are requests for new features, 9% are about problems in the documentation and the remaining 87% are bugs in the code of the several modules.

Around 47% of issues reported are now closed, 33% were not considered a bug, for 3% the decision was made not to fix them, 6% are still open for analysis, and the remaining 11% are in the maintenance pipeline.

These numbers are absolutely astonishing, especially since they come from PHP, with a huge open source community, and considered as one of the most stable and reliable programming language/tools available. Although we only use a fraction of the available functionality, statistically it is very likely that overtime we will use function that have bugs, either solved or unsolved.

My conclusion about this study on PHP is that maintenance is a huge factor even in big, well supported communities. In order to keep IS in production for decades, as we do, we need to improve the way we program computers.

After 60 years of software engineering, almost no software version passes the teenage years, independently of the amount of money, expertise or effort that was put into creating them. Any software project sponsor still faces a new project with a high associated risk of success.

In "Peopleware" [37] the authors of this classic book from 1999 state that "about 15% of all projects studied (…) were canceled or aborted or postponed or they delivered products were never used. For bigger projects the odds are even worse. Fully 25% of projects that lasted 25 work-years or more failed to complete."

The current reference source of success analysis of software projects is the CHAOS Manifest from the Standish Group. The 2012 and 2013 [38][39] reports states that, although it was their best result ever, only 37% and 39% of software projects could be considered a success; 21% and 18% and were clear failures and the remaining 42% and 43% were challenged. The fundamental problem didn't change – building software is still hard and challenging.

However, in small projects – that is, "less than $1 million in labor costs" the failure rate is only 4%. Some argue that the methodologies, paradigms and tools we use today in software

development are fine for small projects, but we still lack the knowledge about how to scale up to huge projects, because of problems that arise in such highly complex assembles.

As time passes by we are getting better at building software, but we are still far from respectful project success numbers when compared to other areas of engineering.

3.2 Other problems in IS

Handling "change" is not the only problem posed to IS. In order to limit the scope of this thesis, without losing track of the holistic approach required to design IS, we will quickly approach other pressing problems of IS design that will not be addressed by this thesis, therefore limiting this work scope.

According to Frederick Brooks [15], "the essence of a software entity is a construct of interlocking concepts: datasets, relations among data items, algorithms, and invocations of functions." This author also states that the essential attributes of software that make them so challenging are: **complexity**, **conformity**, **changeability** and **invisibility**.

Changeability is the main problem addressed by this work, but we would like to add two other problems that we believe are core to IS: **performance** and **correction**.

3.2.1 Complexity

Allen Downey [40] states that "classical models of science tend to be law-based, expressed in the form of equations and solved by mathematical derivation. Models that fall under the umbrella of complexity are often rule-based, expressed as computations, and simulated rather than analyzed."

According to that author, with the usage of computational modeling there has been a shift along the following axis from:

- continuous to discrete
- linear to non-linear
- determinism to stochastic (random, non-deterministic)
- abstract to detailed
- one, two (few) elements to many elements
- homogeneous to composite (heterogeneous)

This new kind of model is often appropriate for different purposes and interpretations of the world.

- From predictive to explanatory

  Many social phenomena's can be explained by simple models like the Schelling's model of segregation, the standing ovation model by Miller and Page's or the threshold models of collective behavior by Mark Granovetter [23].

- From realism to instrumentalism

  Modeling can be a good enough solution for understanding the world and make predictions about it. As George Box said it: "All models are wrong, but some models are useful." [23]

- From reductionism to holism

  Reductionism approach is based on the premise that you can only understand the system when you understand each of its parts. Holistic is the view that some phenomena only appear at the system level and do not exist or appear at the components level. This is also related to the philosophical approach of considering a system in a teleological perspective, that is, considering their use, end and purpose, contrasting with the

ontological perspective that focus on the elements that constitute the system, their interconnections, relations, groups, similarities and differences.

According to Downey [40], these changes may lead to a new kind of engineering and organization of social systems from:

- centralized to decentralized
- isolation to interaction
- one-to-many to many-to-many
- top-down to bottom-up
- analysis to computation
- design to search

And finally, to a new kind of thinking, from:

- Aristotelean logic (only true or false) to multivalent, fuzzy logic
- frequentist probability to Bayesianism
- objective (objectivists – only a single truth exists) to subjective (subjectivists – each person can hold a different truth - or constructivists – truths are established trough social interaction)
- physical law to theory to model
- determinism to indeterminism (breaking the cause-effect link through randomness, probabilistic causation and fundamental uncertainty)

A complex model is not necessarily a better one; actually it is usually the opposite. Albert Einstein said: "Things should be as simple as possible, but not simpler."

3.2.2 Conformity

The problem of conformity arises from the need to interact with other existing systems, and usually not very organized ones.

Systems are used throughout science extensively and by definition systems interact with each other. Systems are normally conceived as something that has an input, an internal structure and an output. According to the ontological approach for systems, as defined by Mario Bunge [41] and Jan Dietz [14], a system has:

- Ambient – a set of elements that live on the border of the system communicating with the exterior (either input or output).
- Composition – a set of elements that live inside the system and only communicate between themselves and the elements in the ambient.
- Structure – links of mutual influence between elements of the ambient and the composition.
- Production – things that are produced by the elements of the composition and delivered to the outside through the elements of the ambient. The production of a system can be a product (material), a result (immaterial) or a service (a mix of material and immaterial goods provided instantly or over time).

Conformity is a major issue in software development. Software developers need to find ways that systematically address the problems of interfaces between systems. Having a well defined but rigid API is not a good enough solution. The most likely solutions is to address this issue through patterns, like the facade pattern, or even better, through pattern languages that combine

typical usages of known patterns to address typical concerns with typical results in terms of non functional characteristics.

3.2.3 Invisibility

The challenge of invisibility comes from the fact that software is an abstraction that is not tangible. The issue of invisibility can be addressed in two perspectives: code view and results view.

On code view, the lines of code expressed in any language does not visually match its execution, therefore it is difficult for the programmer to "see" if the code is working as intended, if it has unexpected unreachable sections of the code, handling all input as expected and where are the bottle necks that prevent it to run faster. When a programmer writes code he does not have a clear relation to any physical objects. Lines of code could be mapped to a visual representation of code in order to make it more understandable and therefore result in software with fewer bugs.

On results view there is the field of information visualization. The process of data encoding consists of classifying data types and choosing the most appropriate visual attributes to represent them.

The first problem is data types. Computer's hardware and usual programming languages typically represent data as booleans, integers (signed or unsigned) or floating points (with a specific precision and range set). This is quite different from the way mathematicians usually classify data (Natural, Integer, Rational, Real, Complex, etc).

On the other hand, statisticians and data scientists typically use only three basic data types [42]: nominal (categorical), ordinal (when we can establish an order between categories) and quantitative. Quantitative can be split in interval and ratio

According to the French cartographer Jacques Bertin [43] there are 7 visual attributes that can be used for visualizing data, for 0, 1 or 2 dimensions: position, size, value (or lightness), texture, color, orientation and shape. Later, these attribute list was extended to 3 dimensions by Card [44].

3.2.4 Performance

The topic of performance in IS can be addressed as the time required to design and implement a solution – the design performance; but it can also be addressed as the execution performance problem, that is – the time a specific computer architecture will take to execute a certain code over a certain data.

Design performance

Programming languages haven't changed much over the last two decades. Adopting new languages is not an easy task because the learning curve is quite steep and the change would only justify if the increase in performance was huge.

There are hundreds or even thousands of programming languages, but most of them are quite similar, and only a few of them are really popular [45] like C, C++, C#, Java, PHP, Javascript, Python, Perl, SQL, Ruby. Even so, there are niche programming languages, for instance for high performance scientific computation the preferred language and the benchmark reference is still Fortran.

A paradigm is an approach to solve a specific kind of problems. Most popular languages support more than one paradigm, typically object-oriented and imperative, or sometimes functional. But very few, and none of the most popular languages, support the four main paradigms.

There any many programming paradigms [46][47], but some are more popular than others: imperative, functional, object-oriented and logic programming.

- Imperative – a program is a state machine and procedures change from current state to a new one.
- Declarative – programs that state what we wish the result to be, not how to get it (HTML, SQL). There are fixed inference rules to get to the stated goals.
- Functional – a program is a set of functions that do not hold any state. Each function returns always the same result based on the same input.
- Object-oriented – objects are closures that hold attributes and associated procedures, typically called methods. Objects are typed to classes and classes have hierarchies that allow sharing properties and methods across those semantic links.
- Logic programming – logic programming is based on statements that establish facts and rules between types of facts. Based on that knowledge it is possible to use inference rules and query for values that are supported by the current fact/rules database. Logic programming is a very elegant way of solving some kind of programming problems.
- Constraint – Relations between variables constraint the set of possible results. The language tries to find out solutions that match those constraints.
- Event-driven – The control flow is determined by a sequence of events, typically human interactions.

The main difficulty in adopting a new programming paradigm is adapting to a new way of thinking in programming logical terms.

Execution performance

Software runs on specific hardware environments. For many years, software guaranteed increased performance just by the increase of processor speed, even without any relevant maintenance over it.

In 2005 a new era started: multicore processors. Processors were not getting faster anymore, but there was still more processing power in the new CPU. Now, increase performance required parallel programming. In 2009, manycore processors (over 16 cores) were introduced, and the production of single core processors ceased at Intel.

The road ahead become clear, but still most programmers don't have a clue how to program in parallel, because they never tried it and were never taught that way. Programmers may know basic synchronization techniques for accessing shared resources (mutexes, semaphores) and communicating through asynchronous queues of messages, but that is not enough to take full advantage of current many core architecture.

Some programmers might have eared about CUDA programming model or the OpenMP standard, but very few ever heard about the Intel Threading Building Blocks (Intel TBB), which has become the best and simplest way to program parallelism in C++ since 2009 when it was launched.

State of the art on programming in parallel is in balancing the core usage for the critical sections of the program and preventing the need for synchronization, instead of handling critical access to shared resources. James Reinders, lead evangelist and director of software development products at Intel says "I am still confident that software development in 2016 will not be kind to programmers who have not learned to «Think Parallel»." [48]

Most algorithms in computer science were not thought to be run in parallel and a lot of work is still needed to optimize them to this new requirement.

Parallel programming is not a challenge that can be solved by compilers, although the great improvements of the optimization compiling phase, because parallel execution is strongly tight with the way we think and write code. Easier compiler optimization problems, like the usage of instructions set for Single Instructions Multiple Data (SIMD), are still not fully used by

compilers, by the same reason, although they were introduced in the processor architecture with the MMX package in 1997.

3.2.5 Correction

A mathematical alternative approach for generation IS is to start with an abstract model of the intended system and then through a sequence of refinement steps deduce the creation of the intended solution. As long as each refinement step is done with the level of assurance of a mathematical proof, the resulting system would guarantee correction. The B method [49] is the most well known strategy that follows this approach.

3.3 Existing approaches to handle change in IS

3.3.1 Agile

For many years software engineering has followed the path of building software as a well organized structure. Due to is complexity, unsatisfactory results and stakeholder pressure to change, software development has been shifting toward agile methods that favor gradual growth instead of the "big design up front" approach.

For most organizations, the development effort is still divided into two phases: development and operation. Even with an agile development phase, the IS is not created to be changed during operation phase, therefore making adaptations more difficult.

Organizations that are natural systems [2] and open systems are more prone to rapid and radical changes and they still don't have the software applications that allow them to better handle their natural ambiguity and in time gradually become better organizations. On the other hand, structured organizations that are based on rational systems, typically have rigid IS which creates difficulties in rapidly adaptation to change and evolution.

3.3.2 Frameworks

Frameworks are software infrastructures that set a stage for the development of applications. They increase productivity but constrain the flexibility of possible solutions by setting a standard way of work for all applications by providing the usual means to quickly solve those common problems.

With strict frameworks, when unusual change requests emerge that cannot be handle by the typical framework structure, specific hacks have to be put in place to get around the framework construction options. In time these lead to inconsistent architectures that are harder to maintain due to incomplete documentation and human resources turnover.

4. Review of Literature

This section contribution:

> Succinct description of the support theories used for this work, analyzing separately the philosophical, ontological, ideological and technological theories.

4.1 Theoretical Framing

Jan Dietz proposed [13] a classification scheme for the set of theories on the field of study of organizational engineering. The arrows among the classes in Figure 6 signify the support they typically provide to each other in order to assure soundness or rigor by being well supported in theoretical terms.

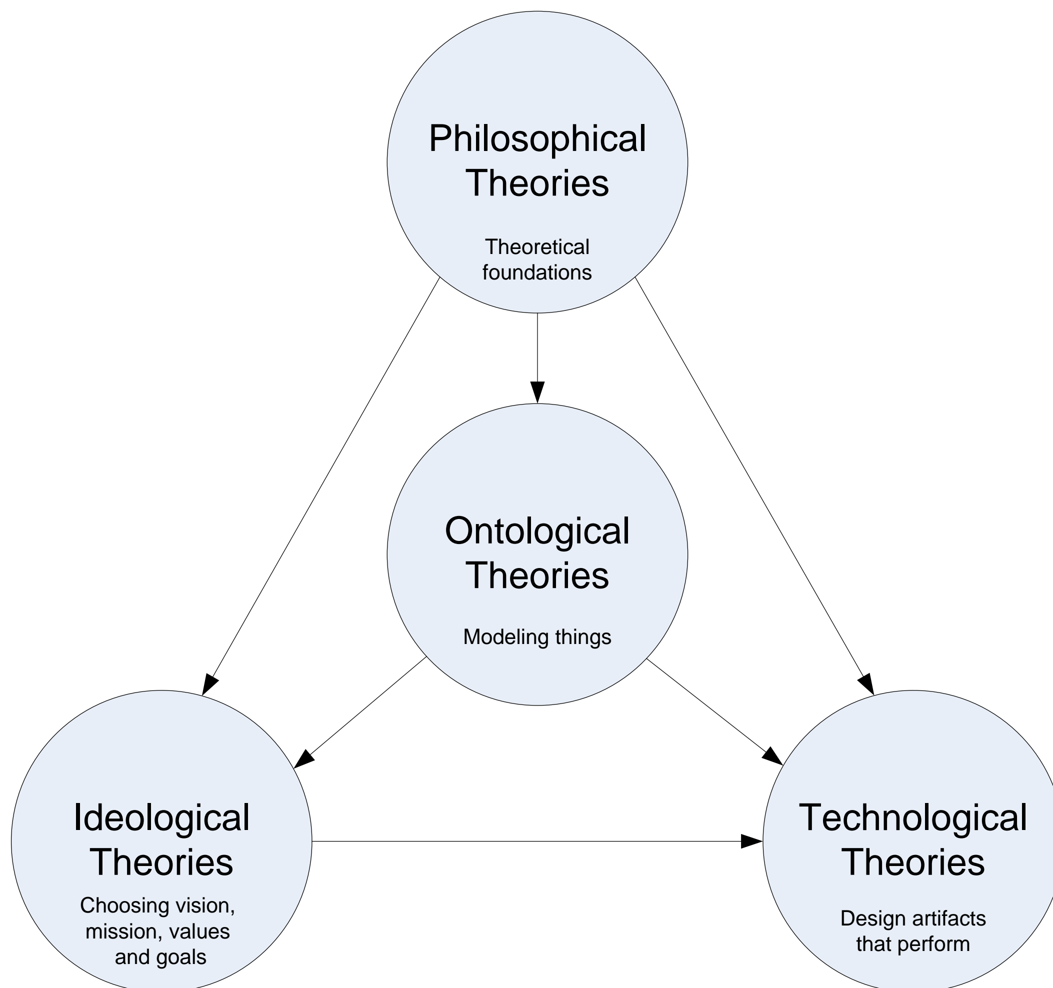


Figure 6 – Foundational Theories for Organizational Engineering [4]

- Philosophical theories concern is the more basic foundational building blocks.
- Ontological theories focus on the nature of things, how to model them, how to explain then, how things relate to each other and use them to predict outcomes.
- Ideological theories are focused on the why we choose to do things in certain ways, based on our vision, mission, values and goals.
- Technological theories are concern with the actual design of artifacts that perform as intended and on the methods to get to those results.

4.2 *Philosophical Theories*

4.2.1 Universal Algebra

One of the most abstract approaches to generic ontologies is universal algebra – a recent field in mathematics that only started in 1933 by Garret Birkhoff [50]. In universal algebra a world can be described with algebras [51]. An algebra is a set of elements of a universe and a set of operations that take some elements in the universe (arity is the number of elements) and map them to an element in the universe, as the operation result.

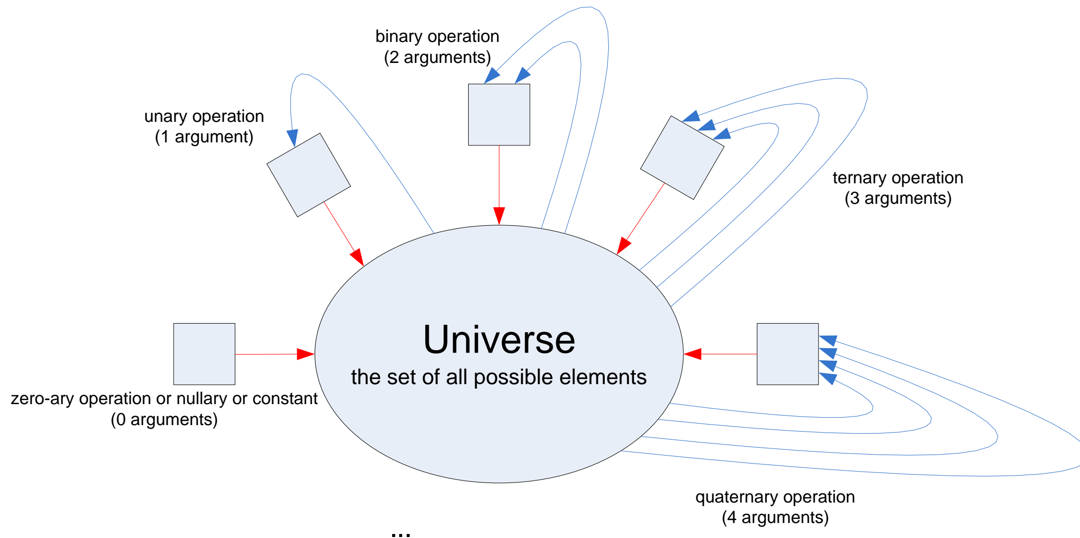


Figure 7 - Arity of operations in Universal Algebra

Each operation can take zero (nullary or constant), one (unary), two (binary) or more elements (n-ary) in the universe and transform them into elements that also belong to the universe. Although there can be any number of input arguments, the output of each operation is always a single element of the universe.

Although these are simple definitions, the elements and the operations can be mapped to anything. The most simple and well known example of algebras are:

a) the universe of natural numbers and the binary operations "plus" and "times" that take two natural numbers and connect to a natural number.

b) the universe with only two members {true,false}, with the unary operation "not", and the binary operations: "and", "or", "imply".

Elements and operations in a algebra can:

a) Satisfy common features like the arity of the universe or the arguments in the operations;

b) Satisfy properties in operations like associativity, distributivity, commutativity, neutral element, absorption element, and many others. All these properties can be expressed unequivocally in mathematical terms and can be verified over real data.

By proving (or checking) which properties apply over an algebra it is possible to name them as, for example: finite, trivial, unary, mono-unary, grupoid, group, Abelian group, semigroup, Abelian semigroup, monoid, quasigroup, loop, ring, ring with identity, semilattice, lattice, bounded lattice, boolean algebra, heyting algebra, n-value post algebra, ortholattice, sub algebra, quotient algebra, free algebra.

Mathematicians underwent significant efforts in characterizing these structures and establishing rules between then, namely: isomorphism, congruence and homomorphism.

Using this tools it is possible to find out relevant properties of the global structure just by knowing which features are supported by each of them. It is also possible to establish a connections between different ways we represent problems, in the same or in very different domains. Knowing these structures that can be treated as the same problem as long as there is a isomorphism between elements, operations and their properties. Using Birkhoff own words: "In achieving this, one discovers general concepts, constructions, and results which not only generalize and unify the known special situations, thus leading to an economy of presentation, but, being at a higher level of abstraction, can also be applied to entirely new situations, yielding significant information and giving rise to new directions." [50]

4.2.2 Set Theory

Set Theory was first established by Georg Cantor in 1874 and has been used since them in nearly all mathematical fields [52].

Set theory uses sets of elements as its core. There are elements of a universe and each set can include that element or not, but there can only be one instance of each element in the set. It is also possible to have empty sets, that is a set that doesn't contain any element.

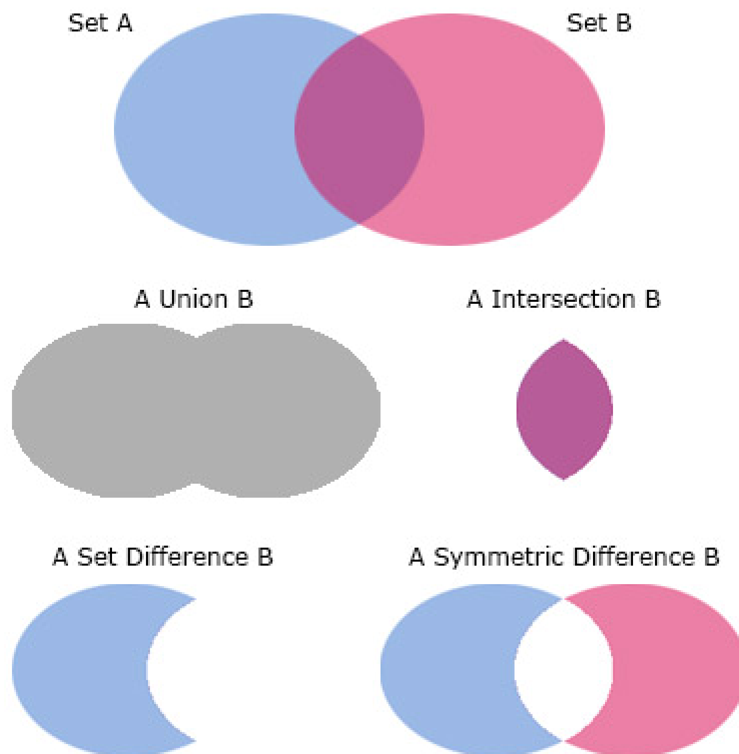


Figure 8 - Venn diagram illustrating some set operations

The usual operations in set theory are:

- **Union**

  Joining all the elements of A and B.

- **Intersection**

  Keeping only the common elements in A and B.

- **Set Difference**

Removing from the A set all elements in the B set.

- **Symmetric Difference**

  Keeping the elements that only belong to one of the operand sets, but not in both.

- **Cartesian Product**

  Pairs with all the possible combinations from elements from set A with elements from set B. For example:

  A={1,2} B={3,4} AxB={ {1,3}, {1,4}, {2,3}, {2,4} }

- **Power Set (unary operation)**

  The result set is all the possible subsets that can be obtained from a set. We can identify a subset by being a set that only differs from a super set by one element. Therefor, the powerset can be obtained by incrementally removing a single element at a time in all possible subset operations from an initial set. For example:

  A={1,2,3}   powerset A = { {1,2}, {1,3}, {2,3}, {1}, {2}, {3}, {} }

4.2.3 Map Theory

Map theory [53] is a rigorously defined formal mathematical theory that extends the Set theory without losing support to any feature of that theory. Map theory includes abstractions for algorithms and metalogic, adequate to support computer science as well as well as being suited to support all classical mathematics.

Classical theories follow the rule "tertium non datur" [53] that a formula is either true or false without any other possibility. Following Kurt Gödel incompleteness theorem, map theory introduces a third and final option: nontermination (or undecidability).

Map theory uses as basic concepts: negation, implication, equivalence (bi-implication). Functions can be treated as black boxes (hiding its content) or white boxes (given more information about its construction). This theory uses ordinal functions to establish order relations between elements as well as other relations between elements; tuples to group related items, rank to assess similar cardinality.

4.2.4 Graph Theory

A graph, in mathematical terms [54], is a data structure that can be represented with four types of elements:

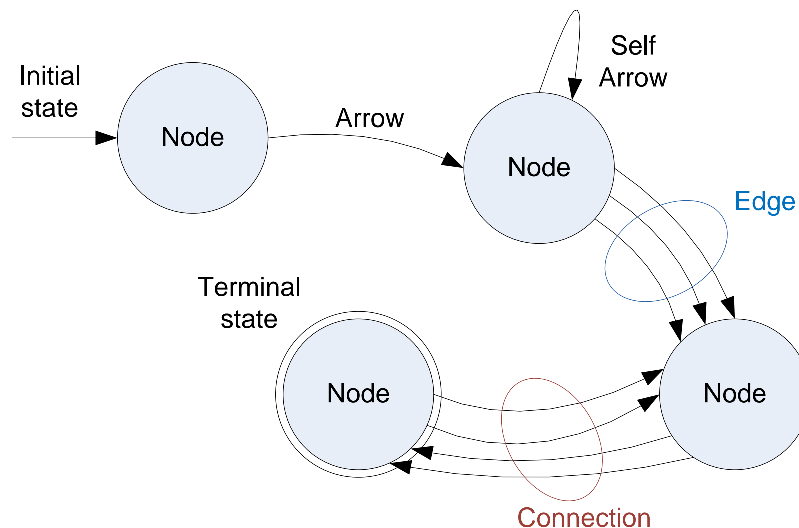


Figure 9 - Elements of a graph

- **Node**

  Node is the core element of a graph.

- **Arrow**

  Arrow is a directed link from a node A to a node B.

- **Edge**

  An edge is a group of directed arrows between the same two nodes (A and B) with the same direction (from A to B).

- **Connection**

  A connection is a group of edges between two specific nodes in any direction (from A to B and from B to A).

Graph theory is a huge field in mathematics that started in 1736 with Leonhard Euler with the famous study on the "Seven Bridges of Königsberg".

Graph theory can use different names to identify its components. In this work we will use these names previously identified. We will call vertices when we refer to arrow, edges or connections without considering directionality or quantity. Sometimes graphs with only directed arrows are called Di-graphs.

It is usual to assign names either to the nodes, to edges or even to each connection from an edge to a node. This is the case of networks of computers with TCP/IP, where each connection gets a unique address, but neither the nodes (router) not the connections (cable) get any identification.

It is also common to assign weights or cost and usage/capacity to each node or vertices. Many algorithms use this information to discover central elements in the graph, paths, circularities, minimum cut to divide the graph in two parts (bipartite). Depending on the rules used to add nodes and arrows to a graph, for example: random or preferential attachment, there can be properties assigned to those properties derived from theory.

4.2.5 State Machines

Graphs can be used to model many things, for example, nodes can represent states, and arrows transitions between states, therefore representing a state machine. Some transitions might be epsilon transitions that can happen without any condition. In this case the state machine becomes a non-deterministic state machine, that is, the current state (node) can be more than one at the same time.

A non-deterministic state machine can be transformed into a deterministic state machine through a predefined algorithm, but in some circumstances that can lead to a deterministic state machine with infinite nodes, which is not possible to be handled by a computer.

Non-deterministic state machines are always more compact than their deterministic counterpart, therefore more comprehensive, even with the uncertainty associated to the current state.

4.2.6 State Charts

A State Chart [55] is an extension of the concepts of state machines and state diagrams for discrete-event systems by adding hierarchy, concurrency and communication.

Although state systems are a natural way of describing the dynamic behavior of a complex system, the combination of all possible states into one "flat" super state model generates an "unmanageable, exponentially growing multitude of states" [55].

State charts introduce the notion of combining several orthogonal state diagrams, creating several layers, that are able to broaden the notion of state diagram decreasing its complexity by avoiding the combinatoric explosion of the number of states.

4.2.7 Data and Metadata

Metadata [56] is usualy described as data about data, but what is metadata for some people might be raw data on which others will build upon to create new data.

There are 3 types of metadata:

- **descriptive** (that tells features about a "thing")
- **structural** (that says how "things" are connected to other "things")
- **administrative** (that tells who has rights about each piece of data and how that data was formed (provenance) and how it should be kept – for how long, how and by whom.

The predicates to be used should, as far as possible, be one of the 15 basic types of Dublin Core [57] (contributor; coverage; creator; date; description; format; identifier; language; publisher; relation; rights; source; subject; title; type) or one of its extensions, in order to facilitate future semantic integration between ontologies.

4.2.8 Basic Data Types

The usual types of data commonly used in statistics are: **nominal**, **ordinal**, **interval** and **ratio** (fixed interval with meaningful zero).

Nominal types are the kind of data that do not have numerical properties, even if a number is associated to each category. Categories are human constructions that are created in the intersubjective world of communities. For example, we can have sex: "male" or "female"; or gender: "man", "woman", "undefined" [58]. The only operations that make sense to perform over nominal data are checking for equality and counting occurrences. These allow the calculation of mode – as central tendency measure – and the variation ratio, giving some notion of dispersion. In order to manage many nominal types, tools should be provided to handle controlled vocabularies, that is, to allow establishing relationships like broader term, narrower term, use for, use instead and preferred term, as is usual in Information Science.

Ordinal types are constructed over nominal types, adding the condition that they have a function that can specify a total order for all members of a set. However the "distance" between those elements is not a measure, since it only specifies the order for any pair of elements, but not the distance between two elements in the group. Examples of ordinal types are Likert scales in questionnaires. Ordinal types can add median to the available central tendency measures and range, inter-quartile ranges to the dispersion variables. Over ordinal types it is possible to construct ranges (with open or closed boundaries) and operations over ranges and it is also possible to perform sorting operations.

Interval types are constructed over ordinal types, adding the condition that all intervals in the scale are equal. For example, the measurement of temperature in Celsius or Fahrenheit is an interval type, but not ratio (as shown below) because zero degrees does not mean absence of temperature. Intervals allows the usage of meaningful differences, but not the application of ratios between those measures. For interval types it is possible to add Mean as a central tendency measure and standard deviation (and variance) as dispersion measures.

Ratio types are constructed over interval types, adding the condition that there is a absolute zero that is meaningful in the scale and that represents the absence of the phenomenon. Examples are measurements of height, weight and time intervals.

For any variables of a certain type, it is always possible to go downwards in the types of variables following the line: ration > interval > ordinal > nominal, but not the other way around.

In order to manage many nominal types, tools should be provided to handle controlled vocabularies, that is, to allow establishing relationships like broader term, narrower term, use for, use instead and prefered term, as is usual in information science.

For ordinal types (and above) it is possible to perform sorting operations. Having a sorted dataset of size N, preferably ordering in insertion and balancing trees with red-black algorithms,

we can get improvements in the retrieval of information, getting log_2N if balanced binary trees are used or log_MN if balanced Bayer threes of size M are used. This would allow fast retrieval even for datasets with 10^{80} records – if we could get them and keep them - which is approximately the number of atoms in this universe (estimated to be between 10^{78} and 10^{82}).

4.2.9 Common Complex Data Structures

There are some data structures that, although complex, are commonly described in literature [59][60][61] as supporting elements for algorithms in most programming languages like lists and trees. Data structures and algorithms are tightly connected and have a crucial role in software development as they allow efficient solutions to be developed. Algorithms that will handle data-sets over millions of items must have linear or log-linear (N x log N) performance, otherwise they won't be able to complete in acceptable time.

According to the author's experience, the following five data structures have the features to allow efficient representation and support for the majority of algorithms: **sequences**, **associative maps, trees, automata, systems**. These data structures can be mathematically generalized to **universal algebras**.

Sequences

A sequence for any finite set of elements can be defined through links that set the order of precedence between them.

The most common sequence is the string of characters. Sequences should be treated as a more general sequence of things in order to reduce unnecessary complexity. In the PHP programming language there are, at least, 98 different functions [62] that perform simple tasks only for strings, but do not work for any other sequence with other type of content.

There are also two special cases for sequences that should be considered:

- Streams – sequences that do not have a known end, and therefore its size is undetermined. Streams must be handled partially when adequate.
- Sequences with partial ordering (not full ordering). This is the case of sequences that include overlapping messages. This means that there is a partial ordering between the elements, that is, some elements have an order between them, but others don't.

If the ordering is not complete, that is, not defined for all elements of the set then it will be a partially ordered set.

This simple model for sequences can be used to model all kinds of queues, stacks and buffers, since there are only small differences in the way that elements are added and removed from sequences.

Associative Maps

Associative maps are typically known as associative arrays and available in many programming languages, like in PHP and Javascript. In this work we call them Associative Maps instead of Associative arrays as we will extend the typical definition for this data type.

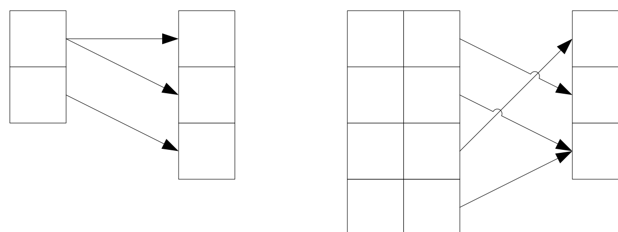


Figure 10 - Examples of Assiciative Maps

An associative map is a structure that links elements from two different universes A and B. Tuples from universe A can have certain arity (one, two or even more), that can be used to map

1D, 2D, 3D or more dimensions. The second mapped elements from universe B typically have arity one.

Associative maps can used to represent a vast group of typical structures.

- A **set** instance can be represented as a associative map with arity 1-1 between some possible elements of a domain into the boolean values representing their presence or absence in the set instance.
- A **bag** can be represented as a 1-1 associative map between elements of a domain (without order between them) and natural numbers including zero, for recording their count.
- An **array** maps a natural number to an element. A **matrix** does the same thing in two dimensions, as does a **cube** in three dimensions or a **n-cube** in n dimensions.

Trees

A **tree** is a special kind of structure that could be mapped as a graph, where each element only has one outgoing arrows (to the parent), and there are no cycles, that is, it's not possible to link in the front to an element that already occurred in the back. In a more general case we have a Bayer tree [63] with a fixed number of N elements (working as an array) and N+1 links to similar structures. This structure is extremely efficient for storing and accessing large amount of ordered information as it allows to grow and shrink and search up to $\log_N X$ performance.

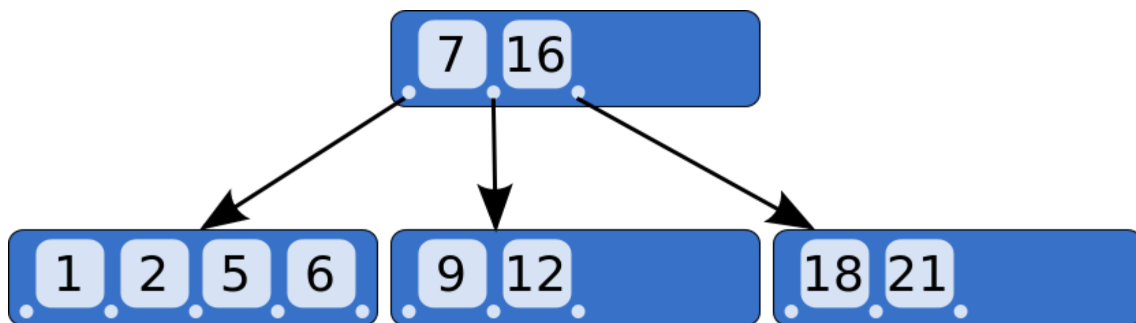


Figure 11 - B-Tree [63]

4.2.10 Automata

A Finite Automata [64] can be used to model state diagrams and grammars for a regular language. Finite Automata can be represented as network, where nodes are called states, with two additional bags, containing the states that are initial and the ones that are terminal. For each arrow in the network there might be a constraint that limits the possibility of following that arrow from one state to the other. It is possible to transform between finite automata, regular expression and grammars consistently.

4.2.11 Systems

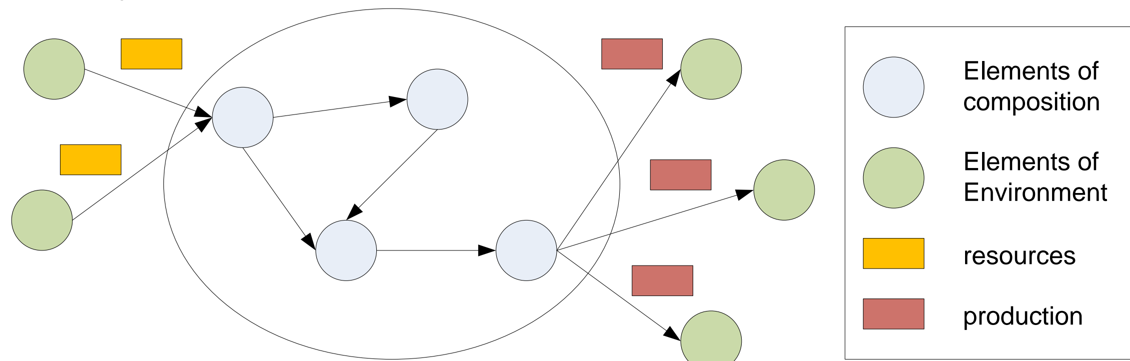


Figure 12 - Elements of a System

Using Jan Dietz definition of system [65], expanded from Mario Bunge, elements can be either members of the composition or members of the environment, and connections are the influence bounds between them. Dietz extended the definition by adding another category of elements, called production, that are the goods or services produced by the composition and delivered to the environment. There should also be added elements called resources that are delivered from the members of the environment to the members of the composition and consumed or transformed into products or services.

The dependencies between the elements of the system are represented using associative maps and sequences.

4.3 Ontological Theories

4.3.1 Performance in Social Interactions Theory

The Performance in Social Interactions theory (Ψ - PSI) [65] was created by Jan Dietz and among other thing rely on a pattern for interactions between two persons: request-promise-execute-state-accept.

DEMO methodology [65] describes the world using transactions. Each transaction can be initiated by a set of roles, but is executed by a specific role. A ontological transaction follows a universal pattern with the sequence of coordination acts request (by requester), promise (by executor), state (by executor) and accept (by requester), complemented with cancellation of previously taken acts in the sequence, which gives a total of 20 possible coordination acts. The executor of each transaction also performs a production act (execute) between the coordination acts of promise and state. Transactions at certain acts can initiate other transactions and have dependencies on other transactions. These dependencies are specified in action rules, one for each act in each transaction.

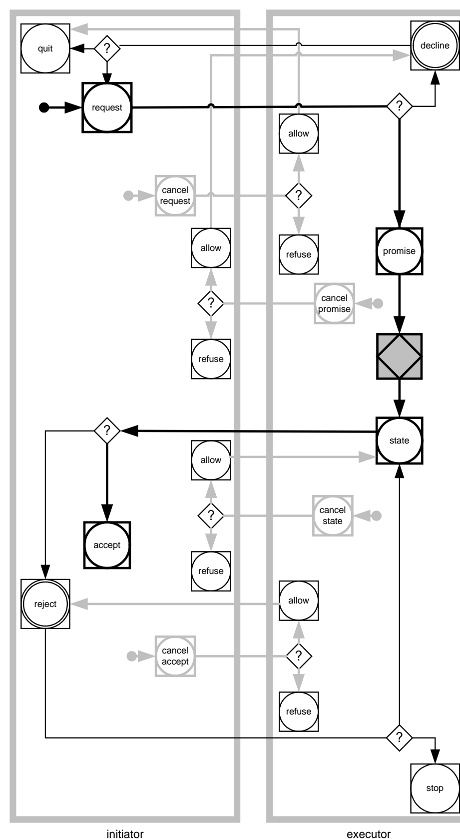


Figure 13 – PSI theory - Social Interaction Pattern [65]

4.3.2 Foundations Ontologies

The common ground of all ontologies is that there are "things" and "connections" between "things" (boxes and arrows), naming them in several different ways and with different meanings. Apart from these simple concepts, no broad consensual taxonomy of things and connections has been agreed upon. As Admiral Grace Hopper said "The nice thing about standards is that there are so many of them to choose from."

Foundational ontologies have tried to grasp the fundamental parts of the world, but tend to focus on one or few particular aspects, like: languages, logic and whole-part (mereology) [66]; processes [67], events [68]; logic and semantics [69]. Even the most evolved foundational ontology, DOLCE [70], states that they "do not intend DOLCE as a candidate for a «universal» standard ontology".

Although these foundational ontologies are substantial philosophical efforts, the majority of their entities lack:

a) the ability to be learned with minimal effort

b) the flexibility to adapt to concrete circumstances

c) the beauty of simplicity

For example, DOLCE taxonomy divides the world in 3 "things" (or entities), in a way that doesn't feel natural:

1) abstracts (fact, set, region, …)
2) perdurants, that is "entities that happen in time" subdivided into events - like achievement or accomplishments - and statives - like states or processes
3) endurants, that is "things that are in time, and their parts flow with them in time", subdivided into quality – like temporal, physical or non-physical - and substantial – like physical and non-physical.

Understanding just this first branch is a significant effort. Although we have studied several foundational ontologies, we haven't used any of them in this work as a solid ground.

4.3.3 Relations Nature

The best structural relations theory known by this work thesis is the one presented in [66] that facing 2 parts (A and B) should ask 3 fundamental Boolean questions:

- Are parts made of the same kind? (green question)
- Do parts have functional restrictions on space or time? (red question)
- Are parts separable? (blue question)

If we map the 3 questions on a Cartesian axis we get 6 kinds of structural relations. Although 2^3 could give 8 options, only 6 are described as possible. The combinations give unexpected results:

- **Stuff/Object** (No No No)
  If parts are not of the same kind, but you are not able to separate them from the result, and if each part does not have functional restrictions, you get a Stuff/Object relation. Examples are: gin-martini or steel-bike. Once mixed you can not separate gin from martini or remove the steel from the bike. Although distinct the parts do not have functional differences over time or space.
- **Place/Area** (Yes No No)
  If parts are of the same kind, but you cannot separate them, and they do not have functional restrictions, you get a relation of the kind Place/Area. You can say, for

example, that Everglades area is an area in Florida, like a specific oasis is a place in a desert. An oasis functions the same in the desert, no matter time or space.

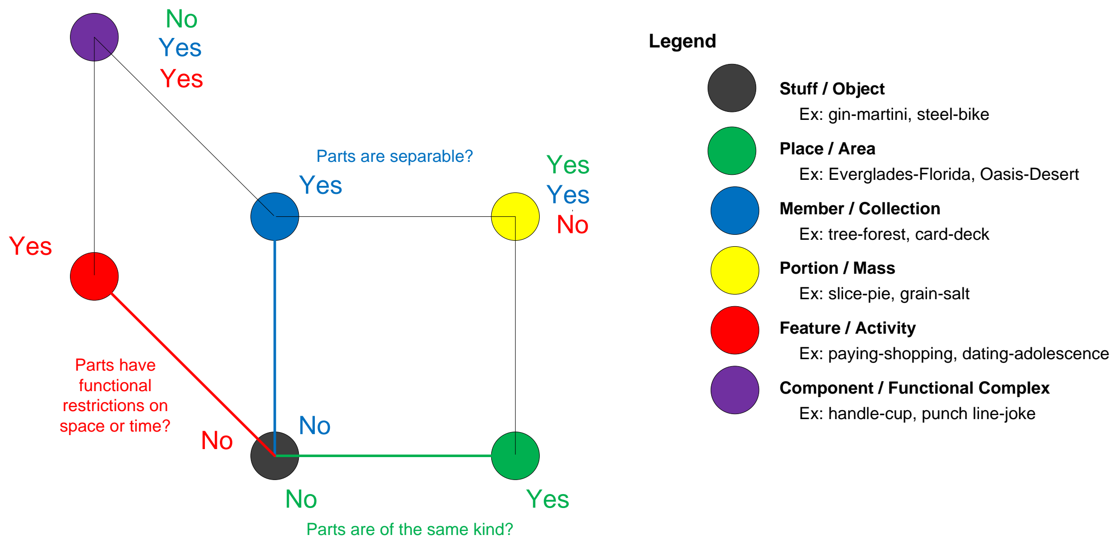


Figure 14 - 3 fundamental questions to determine the nature of relations, based on [66]

- **Member/Collection** (No No Yes)
  If parts are not of the same kind and do not have functional restrictions, but can be separable, then we have a Member/Collection relationship like in tree-florest and card-deck. A tree is not a forest, you can separate a tree from a forest, but a tree functions the same on the forest no matter time or space.
- **Portion/Mass** (Yes No Yes)
  Whem parts are of the same kind but are separable and do not perform a different functional role then we get a Portion/Mass relationship like in slice-pie or grain-salt. A slice of a pie is of the same nature as the pie, but can be separated, but is functional equivalent to the pie.
- **Feature/Activity** (No Yes No)
  Two parts that are not of the same kind but cannot be separable, having functional differences over time and space then we get a Feature/Activity kind of relation.
- **Component/Functional Complex** (No Yes Yes)
  The most generic kind of relationship is the component/functional complex, that you get when parts are not of the same kind, can be separable and each part performs different functions over time or space.

The structural metadata is used to establish fundamental relationships between data. A lot of work has been done in identifying the properties of structural relationships between concepts in foundational ontologies, namely in [72] and [66]. Therefore we will just mention some of its elements and rely further analysis to the references and to further work. In general, structural relations are established between individuals, collections and the universal set of values. Some examples of structural relations are: individual part-of individual; individual instance-of universal; individual member-of collection; universal is-a universal (taxonomic inclusion); universal partonomic-inclusion-of universal; collection extension-of universal; collection partonomic-inclusion-of collection; collection partition-of individual.

4.3.4 Neuron Model

Concepts can be connected with a neuron like structure. In a simple model, a neuron can be represented with a central body called soma, a tree of dendrites that act as input channels and axon as output channel. If we consider each neuron to be a fact, then dendrites could connect to a arbitrary number of who's, what's, where's, etc. Branches in the dendrite tree can have connections that act as suppressors. This analogy is particularly useful because it can model any boolean expression made up of and's, or's and not's, as it has been shown in neural science.

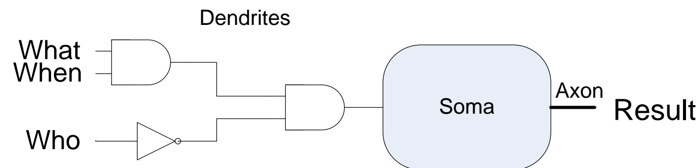


Figure 15 - Neuron Model

Neuron model is particularly useful for data retrieval within contexts – the small worlds in networks. If we have a set of fully energized neurons as our current context, it could be possible to energize the structural elements connected to those neurons (who, what, why, when, where, how). Then it would be possible to compute how correlated are other neurons by the level of energy they are getting in their dendrites, and realizing not only the similarities, but also the differences between those new neurons with the one in the current context. In turns it would be possible to join some of those new neurons to the current context and repeat the process, as long as desired. This model of retrieval of information is, in simple terms, the way Antonio Damásio [21] proposes as the way memory works in human minds.

4.4 Ideological Theories

4.4.1 Business Motivation Model

The OMG group introduced in 2008 the Business Motivation Model [71] that defines the end and the means concepts to structure a vision, goal, objectives, misson, course of action, strategy, tactic, directive, business rule and business policy.

This model addresses the ideological theory concerns by providing a structure to represent the roads to follow and the roads to skip.

4.4.2 Values

Every person has a unique hierarchy of values that may change over time and over context. By repeatedly placing the question "why?" to the reasons for each behavior, you should end up in a core value.

There are many possible values to justify human endeavors, for example: acceptance, adventure, affection, authenticity, beauty, belonging, care, challenge, collaboration, competence, compromise, confidence, contribution, cooperation, courage, creativity, curiosity, culture, determination, development, devotion, discipline, effectiveness, efficiency, empathy, empowerment, enjoyment, enthusiasm, equilibrium, excellence, excitement, faith, family, fellowship, flexibility, forgiveness, fun, generosity, genuine, gratitude, growth, harmony, health, honesty, honor, humble, humor, independence, influence, inspiration, integrity, intuition, involvement, justice, kindness, knowledge, leadership, learning, liberty, love, loyalty, moderation, money, nature, order, partnership, passion, patience, peace, perseverance, pleasure, play, power, prestige, quality, realization, recognition, reflection, respect, responsibility, safety, security, serenity, spirituality, stability, status, success, teamwork, tolerance, tradition, truth, variety, wealth, wisdom.

The set of meanings that are consistently used to add sense to organization decisions create an organizational culture.

4.5 Technological Theories

4.5.1 Normalized Systems

An approach to tackle the changeability problem is building software bottom-up using normalized systems [73]. This theory states that software handles only two technology independent, primitive entities: data and action.

- Data entities only hold data, without any outside clue on the format in which it is stored and without any associated methods (like in object oriented classes), except for the basic getters and setters methods. A data entity can contain structured information like in a structure or record, and can also point to other data entities. All references to data entities have a clear reference to the applicable version.
- Action entities can only contain a single task in normalized systems, that is, it only does one simple thing, although the programmer can decide on the granularity of how simple it can actually be. By separating tasks into different actions we separate concerns. Actions can be hierarchical and include calls to other actions. They use data entities as input and produce data entities as output. All action entities have a clear reference to the applicable version.

The benefit provided by normalized systems is to consider that all data and action entities can evolve into new versions, but in each version the number of changes must be bounded, that is, with limited and predictable scope to other action and data entities. All references to data and action entities always have the associated version. At every time it is possible to know what data types reference what other data types, what action use which data types as input and output, what actions call other actions. This explicit knowledge bounds the scope of any change.

For every new version there is a set of anticipated changes:

- An additional data attribute in a data entity
- An additional data entity
- An additional action entity or version, including but not limited to:
    - have a specific data entity as input, or producing a specific data entity as output.
    - calling a specific action entity
    - using another external technology
    - representing a mandatory environment upgrade
    - containing an additional error state
    - use of alternative algorithm
    - minor change in functionality due to performance or other issues

According to the authors of the normalized system theory, deletions are considered a matter of garbage collection and not a matter of changes to the information system.

According to this theory, any external action, like using a feature of the programming language, framework, package, library, operating service, web service or similar, should always be bounded by a specific action entity. These allow the independent evolvability regarding external systems.

4.5.2 Functions

The most common form of encapsulations that is available in programming languages is the notion of function (or procedure).

In mathematical terms a **function** can also be mapped as an associative array with n-1 arity, that is, linking elements n elements (input parameters), each one of them with a specific universe, to

an element of a specific domain. This is the mathematical notion of function where there is a single result that only depended on input parameters influence the result.

Even in object oriented paradigms, the function plays a major role, and actually calling a method of an object it is almost identical to calling that function passing the object as the first parameter.

5. Options

This section contribution:

> Defines the options for this architecture, by defining the non functional requirements in terms of scope, technology, performance, correction, robustness. control, security, reliability and usability.

Most of the non functional features of a information system are given by their design options. In order to be able to evaluate the design options described later in this document we must first set the terms by which we wish to be evaluated, as there still aren't established agreements on the desirable requirements, neither *de jure*, nor *de facto*.

5.1 Scope and Technology

R01. Have an infrastructure to support the system.

R02. Handle any amount of data.

R03. Handle any type of data.

R04. Handle any number of software applications.

R05. Handle software applications of any size.

R06. Handle any number of stakeholders.

R07. Handle latency and jitter in communications.

R08. Support multi-strategy algorithms.

R09. Support parallel algorithms.

R10. Support multi-paradigm programming.

R11. Support many models to explain and predict events/behaviors.

R12. Support many patterns and their combination on pattern languages.

R13. Remember everything (events) in the system, unless instructed to forget.

R14. Reuse the same code with minimal maintenance cost (build to last).

5.2 Performance

R15. Provide real time response within specified constraints for real time systems.

R16. Be as fast as the supporting technology allows it (including parallel environments).

R17. Continuously improve performance by increase knowledge of system and users.

R18. Allow periodic code re-regeneration based on the new versions of used actions and overtime advanced compilation to memory for finding optimizations in the functions where usage justifies the optimization effort.

5.3 Correction and Robustness

R19. Handle centralized and distributed data consistently

R20. Facilitate the input and output of data with external systems

R21. Support complex data structures in system infrastructure

R22. Support several layers of abstractions to manage complexity

R23. Perform forward inference over events to provide timed analysis

R24. Allow temporal analysis for specified or implicit/typical time ranges.

R25. Allow to travel in time in the system, that is, perform (data and actions) as if the system was in a certain date in the past to perform analysis of scenarios.

R26. Allow to simulate future scenarios based on current knowledge and combinations of minor changes in derived variables.

R27. Allow the system to evolve in bounded ways that minimize the cost of change without "aging" the code.

R28. Assure minimal and identified dependencies with external constraints.

R29. Certify specific requirements in systems or in program code, either by the process implemented or by statistical analysis of runs over realistic scenarios.

R30. Provide means to inspect quality, either automatic or through user collaboration and report them using metrics.

R31. Manage rules, or preferences, that can change over time , and optimize the system to comply with the current rules and preferences.

R32. Allow the robust recovery from a wide range of disaster scenarios.

5.4 Control, Security and Reliability

R33. Allow control actions over registered events and take appropriate actions.

R34. Be automatically resilient and fault tolerant to changes in the environment by automatically trying to find alternative ways to reach the same goal and reaching for system administrative help otherwise (for instance: less memory usage, less communication, less disk space, less parallel processors, faster results with less precise results – in general less resources or less time.

R35. Allow the execution of code either locally or remotely regarding the best interest of the system.

R36. Expand or contract the level of services according to preferences with minimal deployment cost.

5.5 Usability

R37. Allow diverse ways of interacting with the system.

R38. Take advantage of localization to customize interfaces according to the user preferences.

R39. Adapt to constraints and have graceful degradation with increasing limitations.

R40. Run in a new environment with bounded number of adaptations.

R41. Allow to visually check the system and instantly understand its needs and its strong points at each moment in time.

R42. Handle many different interfaces with stakeholders, each of them with a specialized data broker that makes data available to the interface in its proper context, in a timely manner and with the best possible visual language and attractiveness for that user and environment (including accessibility constraints)

R43. Provide self teaching interfaces, at least for the first time users of the system.

R44. Provide multilingual interfaces as a standard feature of the system.

R45. Provide appropriate documentation in many formats using single source methodology and being mostly generated in semi-automatic ways.

6. Vision

This section contribution:

> Top-down approach for the proposed solution, including all the core structural decisions like the global system architecture, called TOPO with its components (persons, interfaces, agents, applications and data cells); the asynchronous message model; the memory system; the transaction protocol; the neuron model with Zachman framework dimensions; future visual programming.

6.1 TOPO Architecture

The global architecture proposed in this work is called TOPO. TOPO stands for Transparent Open Platform for Ontologies.

TOPO aims at being transparent in the sense that it promotes a white-box [14] modeling engineering approach to the development of information systems, that is, a construction combining elements into more complex structures, taking measures [73] to control the combinatorial explosion and the consequent increase of entropy that can arises from that construction as it has to be changed over time [33].

TOPO has the goal of being open in the sense that all elements of its structure are open to be used and improved by the community.

TOPO will be a platform because it aims to provide a set of shared components and a wide range of non-functional requirements to allow the easy construction of a family of applications. Applications generated by TOPO will support social interactions of actors as described in DEMO theory [14].

In TOPO, users and applications will keep the power of initiative to communicate with others whenever they want. TOPO is not a framework since it does not implement the typical inversion of control present in frameworks, by relying on callbacks, and using the Hollywood paradigm [74]: “don’t call us, we will call you”.

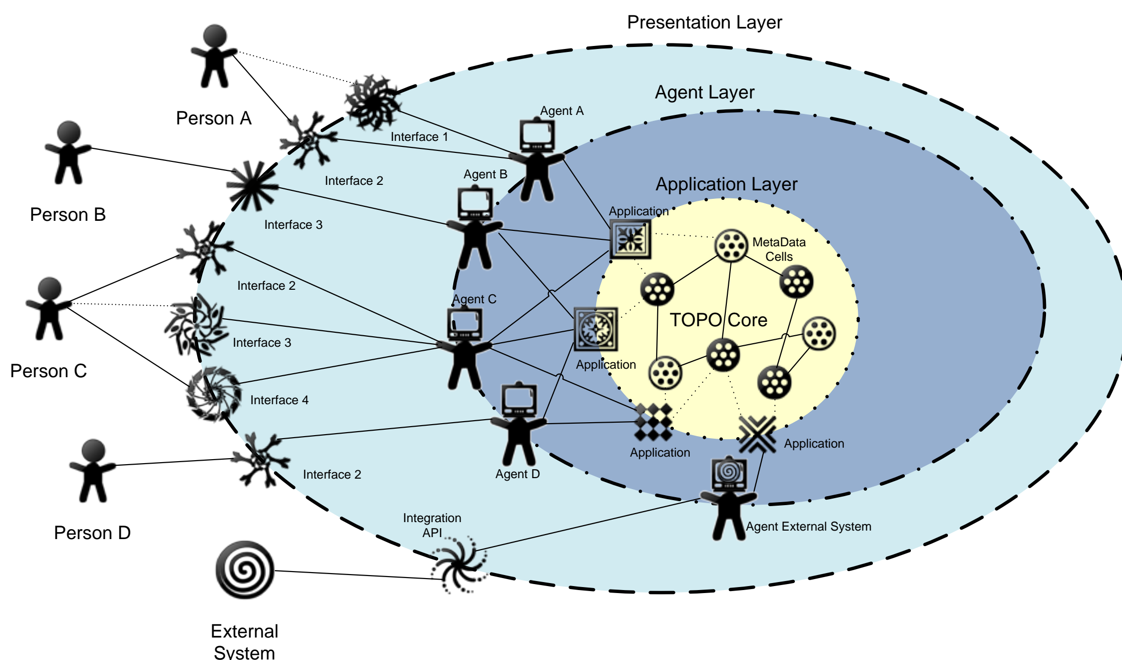


Figure 16 - TOPO global architecture

This new ontology should handle subjectivity, flexibility and the need to handle change in software. By handling change we mean the ability to fully customize generated code over time with minimal code written, but also the ability to get back to the ontological model, change it and re-generate code without losing the previous customizations

The new ontology has to appropriately manage the structural business logic, including both the fundamental core ontological aspects and the operational aspects, using the consistent principles uphold by Enterprise Engineering.

Since TOPO aims to being implemented as an information system artifact, some premises must be stated about the environmental context in which it would operate:

- TOPO is the core of a layered architecture composed by applications, agents, interfaces and persons assembled as can be seen in figure 12.
- Persons can use several interfaces simultaneously to interact with several TOPO applications through an agent. For example, using a smart phone, table, laptop, desktop or wearable devices.
- Each person has a corresponding agent that represents the person in the system, although agents only perform actions in behalf of persons when previously authorized. The responsibility of actions is always on a person.
- TOPO implementation will be user interface independent and all communications with the interfaces will happen through message exchange.
- TOPO implementation will be in the cloud, although interfaces may keep some cached data.
- TOPO will work over the Internet with multiple devices as view/controller, either browser or mobile device interface (e.g. Android).
- All actors will use message exchange as communication paradigm.
- TOPO will have a maximum response time for each API, otherwise promise a response for later.
- TOPO applications will share common data, but can also have private data for each user/application.
- There will be an integration API for sharing data with external systems. The integration will be performed by a specialized agent.

The elements in TOPO architecture are: Persons, Interfaces, Agents, Applications and Data Cells.

6.1.1 Persons

Persons are the main focus of this ontology. In TOPO each user has an agent that mediates his presence in the network. Each person using the system should have a single user, but that might be a constraint too strong for real world applications because persons might require to keep distinct aspects of their lives separate from each other and don’t trust any system architecture to provide that separation. Whenever needed, authentications should be provided by a trustworthy third party matching users in the system to real persons in order to establish contracts with legal validity.

The agent keeps track of the user agenda, that is, the set of purposes, contracts and messages that the user has and had. Persons need to perform social interactions to achieve their goals. To do so they login as a user in the system and they interact with other users/persons transparently using their respective agents. Person identity might be known or unknown by the other part as long as users are authenticated in a trustful way.

For each role a person may perform on the system there are queues and those authorized to perform a certain actor role can view and/or consume the messages in that queue.

Persons need to interact with TOPO using several different devices, and in different contexts of use.

6.1.2 Interfaces

A person can have several interfaces at the same time using either the same device or multiple devices. This requires that a global knowledge of what is the user context becomes available. This is a task handled by the agent.

The growth of wearable devices and the amount of sensors that are now integrated into smart phones, make then formidable tools to be used as remote commands for other devices. Until today, the keyboard was the device with more degrees of freedom due to the number of available keys, but only for expert users. Smart phone sensors, touch screens and device buttons can increase this with more natural interfaces to information systems.

Having multiple output interfaces at the same time, for example several browser windows one on each monitor on a desktop, requires a new model for handling interfaces. There should be several independent interface models that manage the way data is presented in each output channel, working in cooperation.

Also the input channels should be used in cooperation. The use of mouse, touch screen and even other sensors of a smart phone could control other devices besides itself. To facilitate this kind of rich interaction, we should separate each input device and have a specific model to manage its state and input data. Over each input model there can be a gesture detection mechanism that would be able to detect more complex combinations from the same device (for example, CTRL+SHIFT+C in a keyboard). There should also be yet another gesture detector that combines commands from several input devices in several equipments. For example, combining the smart phone gravitational orientation with extra input from the mouse (or the keyboard).

For each task that interacts with the user there is a corresponding interface fragment made of text, images, sound, fields that can be automatically generated based on the required input data, but can also be fully customized to adapt to specific visualization or data input requirements.

Several interface fragments can be combined in a single user interface, depending of the options and constraints of the device being used. A task and the corresponding interface fragment can be shared by several actions.

From the authors experience with the code generation tool, a group of 125 types of user interface fields have been identified with the corresponding view, edit and search alternatives.

These 125 interface fields are of different types:

- Basic fields (like button, checkbox, date, decimal, number, string, time, year)
- Composed Fields (like address, calendar, file, image, movie)
- Fields with special actions attached (like actionlink, currency, datedetailed, email, lookup, lookupfield, lookupimage, lookuprecord, lookupstring, skype, slideselect)
- Fields that keep physical properties with flexible units and convertion rules between them (like acceleration, amount, energy, force, power, pressure)
- Fields than handle layout (like cube, dynamicgraph, dynamictable, gridcomposer, mesh, showhide, space2D, space3D, tabsheet)
- Fields that are managed by the system (like guid, createdby, modifiedby, language, lastsynchronization, tablekey, user)

The full list of types has the following identifying names: acceleration, actionlink, address, amount, area, bag, button, calculated, calendar, category, checkbox, code, color, colortag,

concept, country, createdby, cron, cube, currency, dataquality, date, datecreated, datedetailed, datetimemodified, decimal, derived, dirty, dynamicgraph, dynamictable, effectivedatetime, electriccurrent, email, energy, entity, entityname, entityrelationship, fieldtype, file, force, foreignkey, gendermale, grid, gridcomposer, guid, hierarchy, hook, image, instruction, integer, label, language, lastsynchronization, length, line, linecomposer, link, list, listofcategories, listofchanges, listofimages, listofpointed, listofrelationships, listoftags, location, lookup, lookupfield, lookupimage, lookuprecord, lookupstring, luminousintensity, map, mass, memo, mesh, modifiedby, movie, multiline, number, orderby, orderbybutton, panel, password, personname, point, pointarea, pointer, pointerfield, pointerfile, pointerimage, pointermovie, power, pressure, resultset, scrapping, set, showhide, skype, slideselect, solid, space2d, space3d, spreadsheet, state, string, systemnotes, tablekey, tabsheet, tag, tagcloud, temperature, time, timesheet, untildatetime, user, userid, validity, velocity, viscosity, volume, wavenumber, year.

6.1.3 Agents

In TOPO an agent is a software element that is the representative of a person in the system. Every message sent by a person, whichever interface passes through the agent, as well as all messages sent to a person are actually sent to the agent of that person.

An agent can automate certain tasks when explicitly authorized by its person to do so. Therefore the responsibility of the acts done by the agent are for the person and not any other obscure entity. The agent should accumulate knowledge over time and learn how to better serve its person.

The world is a complex environment where agents interact in intelligent ways to fulfill their persons agendas. Each agent keeps information about their person and about the world from the huge amount of data that the world contains that might interest the person.

An agent can be defined as a software with a combination of the following general set of properties [75]:

Autonomy – The agent can act intelligently, without direct requests from the user, to provide him the best possible service, according to what he knows about the user and the environment and the other known users. The agent does not make any ontological decision on behalf of the user, unless previously instructed to do so, but it can access external data and timely provide/recommend actions to the user.

Reactive – The agent is able to react to changes in the environment, especially if those changes were already anticipated as likely. The agent tries to provide the user accurate information exactly when the user needs them, so that he can make informed ontological decisions faster.

Proactive – The agent proactively acts in pursuit of purposes – the user agenda. The level of intelligence of an agent can be measured by the number of logical actions that the agent is able to conceive in order to reach the user purposes.

Temporal continuity – The agent is always in vigil, optimizing the system and in case of fault, recovering data to the appropriate operational state.

Social Capacity – The agent interacts with other agents in order to perform social interactions. These social interactions are materialized in messages. The delivery of messages to the appropriate agent is a structural functionality of the system. The social interactions follow a strict, but powerful, predefined protocol.

Adaptation capacity – The agent should be able to adapt by learning, that is, change behavior based on past experience on the world environment or about the users, and increase knowledge to improve the anticipation of the future.

Mobility – The agent should be able to move its execution location to another authorized server in the network, either totally or partially, if that is helpful to the quality of service, or if required by the user.

Agents can communicate with each other to create intersubjective knowledge by knowing:

- What we know, what we may know but forgot
- What we know the other know, or may have forgot
- What we think the other know, or may have forgot

Another role for agents is to serve as API for applications, that is, to be gatekeepers of applications, defining how data can be inported and exported from an application. In these cases agents are representatives of the application owner and not the user.

6.1.4 Applications

Applications in TOPO are the information systems for organizations in a broad sense, where organizations can mean any group of persons with a shared goal.

Applications have a set of private data and can also access and share data from other applications within TOPO or external applications through an agent API.

An application is a combination of a Data Model and a Business Model. In the business model are the set of transactions defined for that applications that define what can be done it it. There are a set of common transactions for the normal procedure or organizations, namely the creation, deletion and merge of sub organization unit, the creation of organization roles in each transaction – saying who can do what and also functions.

Functions are groups of people with a certain competence that therefore can perform acts in the transations. There is a flexible mechanism to make a person a member of a certain functional group, as well as assign organizational roles in transitions either to individual persons or functional groups.

The functional groups are particularly useful to handle substitutions when a person cannot perform its role, and also to have a flexible mechanism for delegation, either hierarchical or not.

There are several kinds of delegation:

- **Delegation of the production act**

  doing what has to be done
- **Delegation of the coordination act**

  stating that something is ready to be delivered
- **Delegation of information act**

  informing someone that a production or a coordination was done.

6.1.5 Data Cells

TOPO is a worldwide network of data and users. There is a lot of data available worldwide, but data is not information because the users only want relevant data.

Handling a worldwide data network is a big challenge because users want to access, create, share and modify many terabytes per second, and be able to use part of it immediately, even on their Smartphone’s. In order to be efficient in the usage of data we cannot have metadata in a huge centralized system but distributed in such a way to have the benefits of centralization and distribution, but not an unbearable share of their respective costs.

We chose to use nature as a paradigm of organization and so we conceive the metadata cell as analogy for data organization into small, normalized, modular building blocks. All plants and animals achieve a great degree of complexity by combining millions of specialized cells. Cells differ in size, shape, purpose and function, but they all share tree characteristics: autonomy, self program and communication channels with the outside world.

6.2 Asynchronous Message Model

Messages can be of many types: text, audio, video, symbolic, etc. With an asynchronous message model we can handle either synchronous or asynchronous messages. Synchronous messages are just like asynchronous but with real time constraints, like maximum expected response time.

All messages, no matter the format shall be handled in a standard way using enterprise integration patterns [76] for communication channels.

Messages may pass through several applications and users. The system always keeps track of the location of data and also "provenance" [77][78], that is, the information about who created it, access it, changed or transformed it.

6.3 Memory System

It is becoming everyday more feasible to consider the whole internet as the main data repository and have our local computer disk act as if it was just another cache level like the several levels of cache in our computer memory.

As a general rule of thumb, each level of cache is 8x bigger than the previous. Current computers have disk storage way beyond these usual ratios as it is common to have computers with 4Gb of RAM and 1Tb of disk storage.

The constraint of not having an internet connection could be a real problem if the architecture was not supported on an asynchronous message model, as it would block until a response was obtained. With an asynchronous message model, inaccessible data can be kept in the message queues until the next opportunity of getting an appropriate web connection.

6.4 Transaction Protocol

PSI Theory, presented in section 4.3.1, defines a standard pattern for transactions that are defined around the interaction acts request-promise-state-accept, and the corresponding cancelation acts. We believe that this pattern has 7 problems hereby listed:

1. PSI Theory states that only the requester can initiate a transaction. We believe that both the requester or the executor can initiate the transaction, as long as they perform a two phase commit, so that both know what they are agreeing upon and know that the counterpart also agreed with the terms. In these terms both parts should be treated as equals.
2. PSI Theory assumes that all details of the transaction must be defined in the first request. This doesn't seem realistic, as the requester might not aware of the full range of solutions and details that might be required, but also because these features change over time, and the provider, as a specialist in the field, is in better positions to recommend the best solution for the user without giving him all the knowledge (most of it irrelevant to requester problem) that he has on the domain. We believe that reaching an agreement is a negotiation, where the requester explains the problem that he has, and the provider proposes solutions and configurations. All parts state concerns, adjust the desired solution until they have an agreement or not.
3. PSI Theory assumes that to every request the provider must answer with a promise or a decline. We believe that the provider should have the opportunity to acknowledge the request without agreeing or declining, but instead propose alternatives leading to the negotiation. Therefore, there should be an acknowledge act that does not mean that the request was accepted, only that it was received by the provider.
4. PSI Theory assumes that when one of the parts takes an action the state of the transaction is updated for both parts coherently. Not only this is not easy to assure in an asynchronous world, but also the actions taken by one part cannot automatically affect the other part perception of the state, as the agreement is an intersubjective reality

established by the parts. We believe that each participant acts in the transaction only affect his own view of the transaction. Only when the counterpart acknowledges that act his vision on the transition is updated.

5. PSI Theory defines that there are only two types of acts: coordination (mentioned above) and production. We believe that there are two additional kinds of acts, that each participant can create anytime, and keep it as his own: knowledge acts and meaning acts. Knowledge acts is new information that was discovered because of the transaction. It can happen at any stage of the transaction.

   Meaning acts are internal justifications why certain decision was made instead of another. The meaning acts used for typical occasions build up to define the culture of the organization. Meaning acts are supported by Ideological theories presented in section 4.4.

   Knowledge acts can be generated from production action (learn by doing) or through coordination acts (learn by being) or by other knowledge or meaning acts (learn by knowing). Having knowledge acts associated to the locations that generate them makes a logical way to organize knowledge.

6. PSI Theory defines that each transaction only has one production result that is requested and delivered as a result of a single transaction. We believe that the pair's request-promise and state-accept should be divided into separate transactions. The first transaction should handle the request-promise and the result is an agreement. The second transaction, which is a sub-transaction should handle the delivery of the ontological act, either the decision, or the product that is the result of the transaction. With this division it is possible to have several product deliveries associated to the same agreement transaction. You can even have product delivery and payment as sub-transactions of the same agreement transaction, even inverting the requester and executor roles, as in the case of the payment.

7. PSI Theory determines that each transaction can only have two participants. If more participants have to be involved it is part of the provider to assure their agreements with the terms proposed by the requestor. In reality there are multipart agreements (either multi requestors or multi providers). We believe that in order to have a multipart agreement, all participants must have an agreement transaction to each other participant. To reach an agreement all must agree upon the same terms, and all must receive the terms, acknowledge them, agree with them and the counterpart acknowledges the agreement.

In order to solve the problems presented above a communication state chart was developed to represent a transaction, having 7 possible states for each participant, 14 states in total. The transaction is initiated with both participants in initial state. Both sets of states are identical for each participant, but presented in inverted order for representational convenience.

Instead of the request-promise-state-accept sequence in the PSI Theory, we use a simpler pair of "tell"-"sell" sequence. After each "tell" and "sell" the counterpart can acknowledge the "tell" or "sell" received. A part can only acknowledge a message that was received.

An agreement is reached when one part expressed a "tell" that was acknowledged by the counterpart and after that expressed a "sell" that was acknowledge by the participant that stated the corresponding "tell".

It is also possible to revoke a tell or sell previously stated. When this happens before both parts reach the "tell"-"sell" agreement this happens transparently. When this happens after an agreement is reached then parts move to a discussion state where parts have to renegotiate to reach to a new agreement.

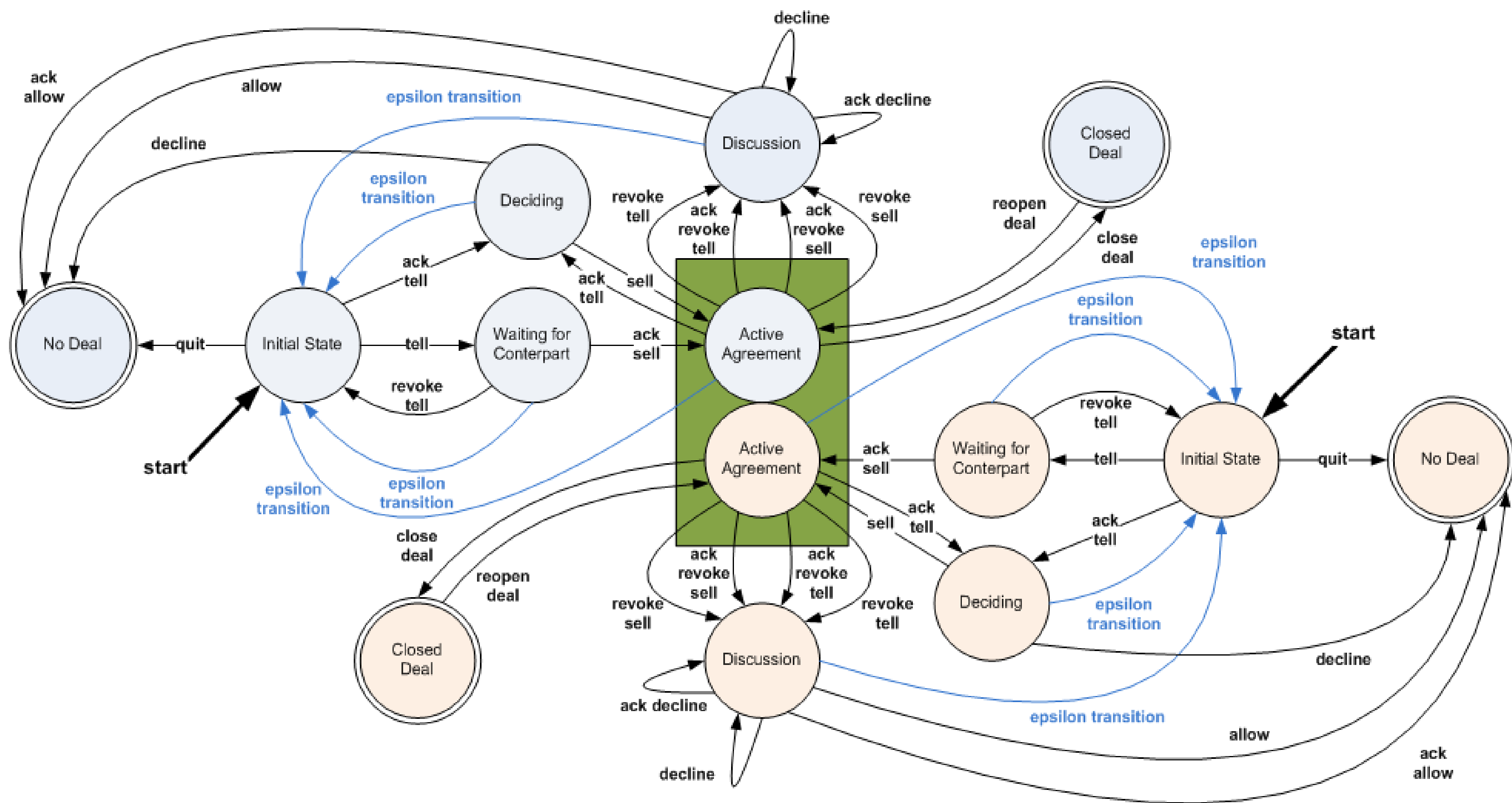


Figure 17 - Communication state chart

If parts have reached an agreement, but after some time wish to upgrade their agreement, with some change to the agreed terms, namely to postpone or anticipate delivery, to add additional features or services, or to specify details that were not fully detailed in the initial request, then they can initiate new “tell” and “sell” sequences without revoking the previous agreement. If an agreement is reached, the new terms are agreed upon and that might require that the old agreement is either closed or marked as no-deal and be replaced by the new one.

Sometimes the amount of discussion between parts may be so big that parts may require to restart the transaction with the current negotiation phase, so that information is clearer about the agreement at stake. Giving an example from a software development transaction: Sometimes, the amount of change is so big, or we are starting a new cycle in the spiral model [79], or a new sprint in the agile [80] principles, that although we are in the same transaction, eventually with pending issues from previous iterations, it is as if we were starting over.

Delegation consists of someone with the authority to assigning the responsibility to someone else in the organization or outside, the ability to perform coordination acts in his own behalf. Delegation should be specific about the acts being delegated. It can happen to delegate any combination of acts. – promise, promise+execute, execute+state, state, execute+cancel promise, etc. By definition, the protest act should not be delegated to the same person that has the rest of the delegated actions, but in some case it may happen. Some transactions can have special delegations configured to be performed as default actions after others.

Authority can also be delegated, but only to someone inside the organization. Authority and responsibility follows predefined hierarchical chains in organizations. Sometimes delegation is accompanied by directives either specific or generic. Those directives or specific decisions that generate doubts can create questioning acts. There might be requested that exist datalogic acts informing the delegator of advancements, or not, for each type of act. These rules can change as time surpasses certain thresholds.

Avocate – Everyone that has the authority to delegate, has the authority to avocate, that is, restore the responsibility of performing the previously delegated act (either explicit or by default).

Although there are new coordination acts “tell” and “sell”, the basic pattern request-promise-state-accept can still be fully maped by this pattern.

All coordination acts should be configured in order to be able to have a quarantine period (be default per user or by request). When this happens datalogic and infologic acts can exist within the organization notifying the yield result and the quarantine status, allowing internal stakeholders to question decisions. It should also be possible to setup alarms that check for specific issues (total price, total amount, etc).

There should also be other infological and datalogical interactions to query to current status and estimates of completion, and specially to get a yield, that is, the current most likely end result of this function, whenever possible. Knowing what is the most likely result, without being certain of it allows stakeholder to either advise against before decision is taken, protest after the decision is taken, prepare for the result in a more predictable date. All infological and datalogical information’s about yield are not definitive and no one can take them for sure, but people can prepare for the most likely result speeding up the process.

All production and coordination acts should be logged in the system. We believe that production, coordination, meaning, knowledge facts are the core that makes the state of the world. The current value of a specific variable is just a derived fact. Of course it is useful to know the current state of the derived facts. We should keep track of it hour by hour, day by day, week by week, month by month, etc. And then we should analyze those derived facts to generate more derived facts, like the ones typically associated with data mining. Basic facts can never be forgotten, but derived facts should be forgotten after a predefined period depending on the importance of what they handle. Using aggregation functions over data, we could slowly but consistently look for association rules, co-occurrence, basket analysis, clustering, classification,

regression, test of models, neural networks and any other machine learning technique we can grasp on. It might be slow, but if it is consistent we could get there.

6.5 Neuron model with Zachman Framework Dimensions

Zachman Framework dimensions – are the elements (who, what, why, when, where, how). These elements are very commonly known as they are the basis for the construction of news.

In TOPO, a "who" can be a "person", an "organizational unit" or a "role". A "role" is a generalization of a "person" to allow it to be performed by more that one person over time. It can also be seen as a function in the organization, and have several persons assigned to that function. "Organizational units" can be arranged hierarchically with great flexibility using the "part-of" structural construction. A person with authority to do so, can link a "person" to a "organization unit" ("part-of") or a "role" ("instance-of") with certain mandatory start instant and eventually a end instant. It is also possible to link an "organization unit" to a "role" with the operation "instance-of".

Elements in "what" are the most complex and versatile element in an ontology definition. Here, artifacts are created and structural relationships are established in order to construct systems, data structures, languages and whatever is required to model the intended application. Using the meaning triangle [14], "what" elements can be categorized as symbols, objects or concepts and establish links between them.

A "why" is a tree of "reasons". For each reason we repeat the question "Why?" until nothing else can be said other that a fundamental value that does not need further justification. Therefore, the "why" can be represented as a three where the branches have "purposes" in free text, and each leaf is a "value".

In the modeled processes of an organization, the values are builtin on the construction of the systems, and do not need to be expressed while in operation. However, when unexpected things happen and decisions have to be taken, values can help to choose the right track among different courses of action, according to what the company has set as their core values and strategy, and also according to value conditions or restrictions that influenced process design at the respective organizational change context. [22]

The dimension "when" handles time references (moments and recurring periods) and time ranges. Time is one of the most problematic topics because society does not use a good model to handle time, since we have multidimensional layers with great level of ambiguity and inconsistencies. Even considering only the western world calendars, we have years with different durations (in days); complex month configuration; months that are not multiple of weeks; flexible timezones; hours in base 24; minutes and seconds in base 60; twice a year time changes to summer time or winter time creating duplicate time references or one hour of unexisting time (which can also be a problem). The solution of having a multidimensional data set with configurable dimensions seems to be the most reasonable solution. As basic configurations the following dimensions should be added: century, decade, year, month, week, day of week, day, hour, minute, second, millisecond.

In order to manage these complexities, TOPO has taken the following options: a) All time references use a local time reference; b) Time can be expressed by persons with negative numbers for times before the reference moment; c) Reference dates (for example: Easter or week 10) can be referenced for any year; d) All time moments have a granularity and always refer to an interval of time. When referring to persons, the minimum reference time is 100ms, but for computer activities it should be smaller but configurable, according to the requirements of the real time systems that are target of design and implementation.

The dimension "where" is a controlled vocabulary that can reference to many spaces, either in absolute terms of in relative terms. Unlike time, there are very well established systems for coordinates (cartesian, polar) and available tools to use these systems, even in mobile

equipments. A location can be set using a global coordinate system (absolute) or a local one using objects or earth magnetic field to establish coordinates.

The dimension “how” can have multiple interpretations, either how to transform an ontology into a working application (the missing work to be done in engineering terms); or in a functional perspective of social decomposition of activities.

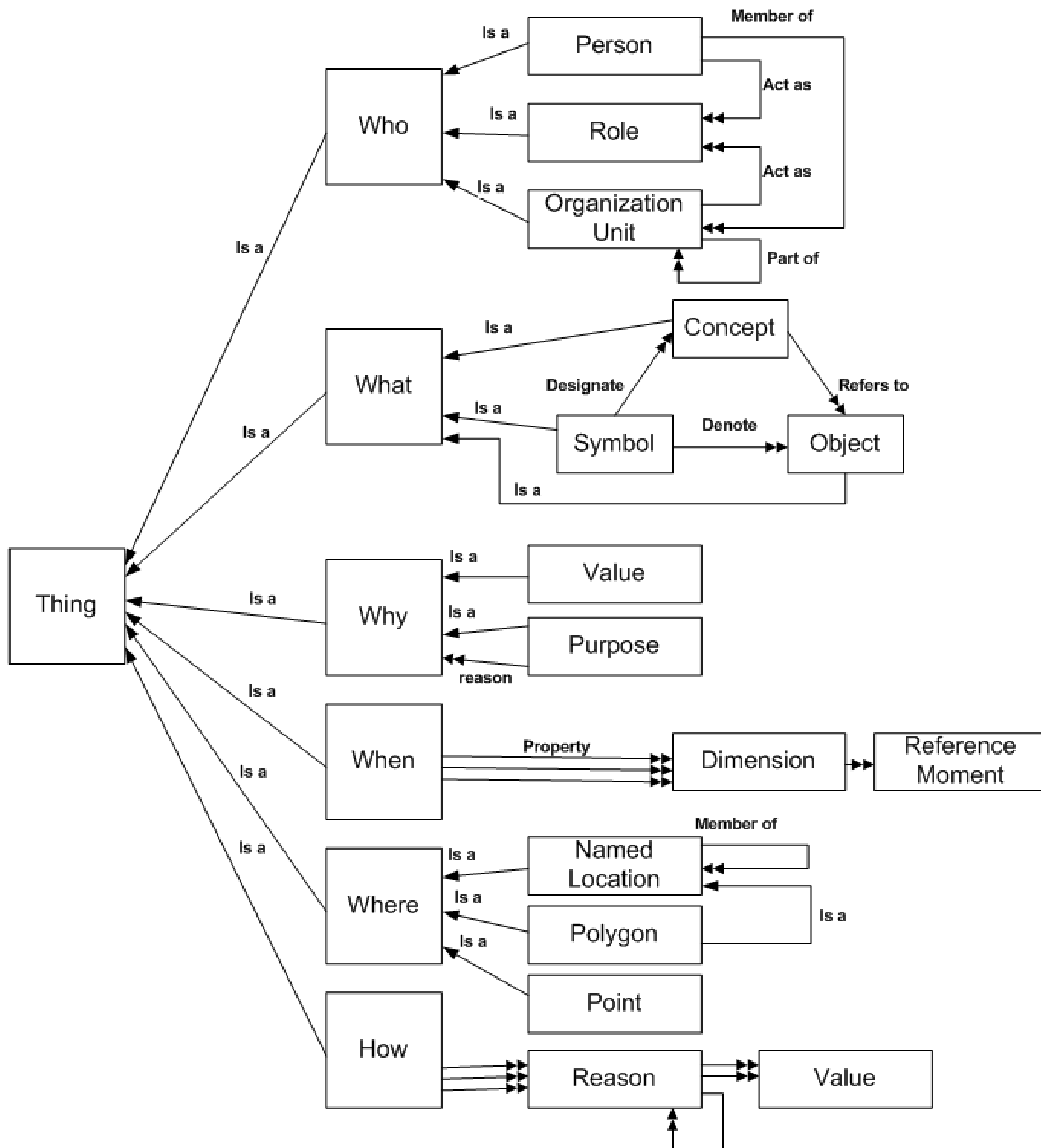


Figure 18 - Structural elements of Zachmann Framework

6.6 *Future Programming*

It is my conviction that it will soon be possible to build on the existing compiling techniques to allow programmers to code their algorithms in the language they prefer and integrate that code seamlessly with code done by other in different programming languages and with different programming paradigms.

Graphical Representation

With the exception of event driven and interface driven languages, most programming is still based on text. Not only text but text a lot of combination of special characters (!= { } [] $ & && || >= % , ; : .) that slow down programmers and make every language a syntactic and semantic nightmare for novice users.

We believe programs should be expresses as graphically as possible. Typical control flows and goto's are visually appealing and, with the proper editing tools that handle blocks of code logically and in a collapsible way, code could be visual and easier to write and understand even for novice programmers.

Control Flow

Some of the newest languages include or added support for new types of control flows over data, allowing easy filtering and iterating options. Selection, filtering and manipulation options like the ones provided by the JQuery extension on Javascript, or PIG relational algebra for advanced queries over map-reduce on Hadoop, or even the ones provided by default on the Ruby language can really increase performance by easier manipulation of data.

Many years ago a claim by Edsger Dijkstra against goto instructions [81][82] give rise to the omnipresence of the structured paradigm and its typical flow control structures: if, elseif, case, while, do until, for. Goto instructions are at the core of the assembly instructions, therefore they are a natural way of optimization and when used locally, inside small functions, with a limited set of possible destinations, and especially if we are using a graphical notation as programming language, goto's can be helpful instead of harmful. We believe that some code is much more easier to express and understand that with unnecessary additional levels of nested structures that control structures sometime force us to use.

Flexible Types

We believe data elements should not be strongly typed. Or at least not by a single type. A type can mean a lot of different things – format, constraints, authorization to be used in some function calls, etc. We believe data elements should have the autonomy to manage their information in such a way that could match many types of data automatically and complementary. This extends to object oriented and inheritance. These semantic relations could be implemented without constraining the amount of different classes an object could be associated with. For example, a horse could be an instance of animal, and also an instance of transport and also an instance of athlete. For each kind of class the data element would be able to respond appropriately and be used in the appropriate context.

Grammar Extensions

We also believe that languages should facilitate the inclusion of grammar extension, allowing future evolvement of paradigms in an easier why. This feature could also allow control structures to be written in the programmers own language. Another improvement to help novice users for the programming language would be to allow a short version based on symbols and a longer version based on words, lowering the barrier of semantic understanding of written code.

Code Visualization

When a programmer writes code he does not have a clear relation to any physical objects. Even so, we have registers, used memory, used disk, time, sequence of code instructions, alternative input values for a function call, etc. All that could be mapped to a visual representation of code.

We believe we should taking advantage of human vision by using images (or moving images - movies) to make software tangible. Vision is the most powerful information channel to the human brain [83] because 70% of human sense receptors refer to vision. Humans are used to handle vast amounts of data through vision – typically a computer screen has more than 1 million pixels, each of them with up to 24 bits of color codes, but it can handle it and we are

able to focus on very specific areas when needed or just get a holistic perspective of what is being seen.

Human vision functions as a massive parallel processor that is capable of preattentive processing, that is, independently of the number of distracting element in the image, there is a pop-out of certain basic visual properties in less than 200-250ms, while the eye movement takes at least 200ms. These notifications are done in parallel by our eyes "hardware" even before we decide to move or focus to interesting aspects of the image. According to Healey [84], there are many properties that share this feature: length, width, size, curvature, number, terminators, intersection, closure, color (hue), intensity, flicker, direction of motion, binocular luster, stereoscopic depth, 3D depth cues, lighting direction.

Position and size can be used for any of the three basic data types. Value (or lightness) can be used for nominal or ordinal, and perhaps also for quantitative, but more challenging because of the difficulties of distinguishing between values. Texture, color, orientation and shape can be used to nominal data. Texture can sometimes be used for ordinal data as well, but also more challenging.

In spite of these difficulties, it is important to realize that information visualization of code execution can also be very powerful. With the power of images we could be able to see trends, find inconsistencies, watch for outliers, elicit patterns, detect changes in behavior, control modes, discover inefficiencies, tackle errors and many other features that are simply not visible from the boring text format in lines of code.

The eyes are not like cameras. Cameras have good optics, single focus, white balance and exposure. They capture the image as a single thing. Eyes, on the other hand have relatively poor optics, they are constantly scanning (saccades), adjusting focus, adapting white balance and exposure and lead to a kind of mental reconstruction of the image, that is quite different from the real image.

We also have to consider the bias our senses have. According to Stanley Stevens power law [85] humans have a good perceptual estimation of length (.9 to 1.1), but systematically underestimate in area (.6 to .9) and even worse in volume (.5 to .8). This means that, although counter intuitively, we could improve magnitude perception by acknowledging the bias and using it in favor of effectiveness of perception by artificially scaling up area and volume instead of using absolute scaling. Accuracy in magnitude estimation in humans is the following (in decreasing order): position (common) scale, position (non-aligned scale), scale, slope, angle, area, volume, color (hue/saturation/value).

We believe we can automatically generate dynamic 3D models from the current code as a compiling phase. With these models it would be possible to communicate in a clear and effective why, without losing the consistency and integrity of the code being represented. It would also be very stimulating for the programmer to look at code and discover much more than was stated in the lines of code in an appealing way.

Most debugging activities rely on peeking certain variables at certain locations to check for their state. Unfortunately, due to the nature of code, bugs can be introduced even on those debug activities and, like in the Heisenberg uncertainty principle, change the way the code is executed, memory allocated and duration of the code execution, especially difficult if it is executed in a concurrent environment. With code visualizations it could be easy to view outliers that do not flow the desired stream of execution because of buggy code or unhandled input data scenarios.

The advent of parallel computing, where multiple cores execute multiple threads or map-reduce operations, creates an even harder unnatural context for our brains to detect bugs in lines of code, therefore leading to more challenging programming tasks and more likely occurrence of errors.

It is currently possible to generate scripts for tools like 3D Studio that generate geometric realities, either static or dynamic. After the model is created, it is possible to navigate in it and look at details. The purpose, like in building blueprints, would be to finding bugs, reveal

inconsistencies, bad design options and suggest possible optimizations, not only by looking at specific cases, but also by looking at the general picture.

As Cristopher Alexander [86] shown in the classic “The Timeless Way of Building”, our cities are made of recursive patterns. In time, patterns join together in repeating combinations creating a pattern language [87]. In my opinion this is exactly what happens in software programming, and we believe that is the key to tackle software complexity.

The prevailing model for materialization of software is still the old Turing Machine, that is, a device that handles symbols on a strip of tape according to a table of rules. All theoretical computing still uses this image as model because it is a simple model and the results, apart from efficiency issues, are equivalent to a modern computer.

Modeling software in a physical structure could be considered in small chunks, like a function at a time. In time it might be possible to get a broader picture, at a different level of abstraction, with hidden complexity. The complexity of trying to represent everything in a single 3D model would be the equivalent of having a single blueprint with all buildings in detail of an entire city.

- Imagine that you are able to model computer components as areas in a surface of a 3D model. You could represent the CPU registers, the graphic processing unit, the memory, the storage space in disk, the bus and the network connection with the outside world.
- Imagine now that as code is run the code you are able to see memory space or disk space being allocated and dealocated by rectangular shapes rising or collapsing in specific memory locations. Imagine that those boxes change color according to recently of usage (read or write) and that the height of the box grows with the number of uses (which might be different for each inner block of the box).
- Imagine that the code flowchart is expressed graphically and that every possible combination of input values (or a good statistical sample) is expressed as a sphere. Then the automatic script should animate the spheres to map the execution of the code instructions. It is my conviction that when watching the resulting movie, concealed patterns would emerge as something different, and those would only be noticeable in physical structure as the one we suggest here. As Aristotles realized so many years ago, many times, “the whole is greater than the sum of its parts.”

By observing the patterns for the input states it would be possible to create clusters based on behavior and that would be a great source for code optimizations that, using the strategy pattern [88], could mean having preferred alternative code sequences depending on the function inputs.

It would also be possible to perform reverse analysis for each cluster (or optimistically for the all input/results pairs at the same time) and through the usage of data mining techniques discover new rules of association between the output and the input, that would allow the creation of a new code alternative to be added the strategy pattern that would produce the same result but with a more efficient code. Probably the resulting code would not be comprehensible to humans, just as it happens now with compiled code. But the original inefficient code, but readable by humans, would still be stored in the strategy pattern for future modifications and the process could then restart.

Code Complexity

Current analysis of code complexity are based on variations of the cyclomatic complexity or the big O notations, that handle only the average case (or the worst case scenario) and not the actual results based on function’s possible inputs. An even better optimization would be to optimize code based on real usage inputs, leaving the unusual input combinations as a fallback in the strategy pattern.

We believe that all functions should store their input and output results in a log, as previously mentioned, not only for regression testing when a change is made, but also to make this kind of optimizations possible in the near future. Notice that, until a bug is detected and fixed on software in business operation, many inconsistent states in the data model might have been

produced. Without an easy way to find those potential inconsistent states many other bug might emerge in cascade as time goes by and they will be very hard to track because the cause/effect link would be lost in time.

Programing languages typically have several primitive data types like booleans, characters and several variants for representing numbers. We believe that all these should be modeled as concepts.

A possible way to tackle the changeability problem is building software bottom-up using normalized systems [73]. This theory states that software handles only two technology independent, primitive entities: data and action.

Data entities only hold data, without any outside clue on the format in which it is stored and without any associated methods (like in object oriented classes), except for the basic getters and setters methods. A data entity can contain structured information like in a structure or record, and can also point to other data entities. All references to data entities have a clear reference to the applicable version.

Action entities can only contain a single task in normalized systems, that is, it only does one simple thing, although the programmer can decide on the granularity. By separating tasks into different actions we separate concerns. Actions can be hierarchical and include calls to other actions. They use data entities as input and produce data entities as output. All action entities have a clear reference to the applicable version.

7. Existing and Proposed Solutions

This section contribution:

> Bottom-up approach with comparison between existing and proposed solutions to implement the architecture, from the most basic data types to the most complex data and action structures. Also an eight step method to allow organizations to gradually move up the formalization ladder in order to improve their information system adapted to their needs.

7.1 Bottom-Up approach

7.1.1 Basic Data Types

Programming languages share a common set of typical data types: boolean, integers, floating points, strings, date/time. Except for Booleans, all these types usually have variants based on their representation size.

Integers

Current CPU instructions assume that data can be stored in bytes, words (2 bytes – 16 bits), double words (4 bytes – 32 bits) and quad words (8 bytes – 64 bits). Over these size specifications there are the alternatives of storing numbers as signed or unsigned numbers (reserving a bit for representing negative numbers) and also an uncommon format called Binary Coded Decimal (BCD) that uses 4 bits to represent each digit in decimal base, that is, only using 10 of the 16 (2^4) available representation power.

For integer numbers over 2^64-1 (~=1,8E19) there is no standard way of handling them in assembly and usually this responsibility is transferred to the programming languages – that just creates an exception, an error or ignores it.

These ways to represent numbers are not natural to users – that just think of them as an infinite set where the smaller numbers are used more frequently. Actually, most of the times numbers are being represented as floating point without we even notice it. This limitation is quite real but unknown by almost everyone that uses computers. You can try this in a simple way – open your favorite spreadsheet (Microsoft Excel, Open Office Calc, …) and write =2^50 in a cell and =2^50+1 in another one. Subtract the two cells and don't be surprised if the result gives you a zero… Now, try the same thing with =2^49 and =2^49+1 and you will likely get a difference of 1. This simple example show how easy computers can produce silent errors in custom applications with big numbers without even giving a warning to users. Some might argue that 2^50 (~=1,3E30) is a huge number, but it's not that big depending on what you are working on. For instance, permutations of 100 in groups of 20 ($\frac{100!}{(100-20)!}$) or even a simpler expression like factorial of 30 (30!) is bigger than that threshold.

Floating Points

Floating points are ubiquitous in computer systems, either in CPU or in specialized units like the Graphic Processing Units (GPU). Most programming languages equally have a floating point data type.

The CPU has several alternative formats for floating point numbers with different precisions [89]. All formats for representing a floating point share a common feature – the existence of a rounding error - you can only have a fixed size of digits of precision, either in binary base or decimal base. For storing a floating point number you need a sign (+ or -), a fraction with a predefined size (limiting the precision) and an exponent of a certain base. Due to this kind of representation numbers between -1 and 1 have a lower representational error, but as numbers get bigger the associated error increases.

The most recent version of the norm about floating point number [90] is IEEE 754 (August 2008) that only uses base 2, but there is also the IEEE 854 that allows base 2 and 10. These norms state the various alternatives of representation the floating point numbers, but they also set the way to handle infinites, NAN, rounding rules and exception handling like division by zero and overflow.

Every time an operation is performed over a floating point, even a basic operation, the uncertainty error associated with the result increases, that is the maximum difference to the real number (above or below) that the result has.

Every time a sum or a subtraction is performed the boundary becomes equal to the sum of the uncertainty error of the operands.

Every time a multiplication is performed axb, with "ea" and "eb" as their associated errors, the uncertainty error is multiplied by $(1 + \frac{ea}{a} + \frac{eb}{b} + \frac{ea\ x\ eb}{axb})$. When a division is performed (a/b), in fact two multiplications are performed – one to get 1/b and another for *1/b x a* - increasing the error associated to multiplication.

In order to minimize the error the order of operations should change whenever possible. We should always perform addition and subtraction first, and then multiplication and only the latest possible time division should be performed. Performing many floating point operations in sequence always increases the boundaries of the associated error, and we are constantly doing it without any concerns about it or even a notion of the order of magnitude of the error we are introducing.

Due to this associated rounding errors, even the most basic operation that is the equality test cannot be performed between two floating points because equal numbers can be stored with associated errors that make them different. This representation method also allows that the same number can be represented in more than one way.

Strings

In the old MS-DOS, the first wide used operating systems for the 8086 computer architectures, characters were represented with 7 bits (128 options) that included letters a-z, A-Z, digits 0-9, and all the custom symbols on the keyboard and some extra ones like null (0), start of heading (1), end of transmission (4), acknowledge (6), backspace (8), new line (10), carriage return (13) and many other eccentric "characters" numbered in the lower part of the chart up to 31.

Then appeared the need to use computers in countries that did not use English and ASCII pages were extended to 8 bits (256 options) and alternative extension pages were created.

There was also the emergence of new symbols like the euro (€). Although UTF-8 was defined in 1992, its adoption only came many years later. UTF-8 is now the most common encoding on web pages, but are also the utf-16 and utf-32 alternatives…

UTF encoding have the advantage of multiple size encoding, so common characters are represented with fewer bits than the most uncommon ones. There is still no established unique coding standard and different encodings is a huge source of problems. It is almost certain that these encodings standards change in the next decade.

Several encodings are still a problematic issue that creates compatibility problems. For example in a MySQL Database there are 195 collation schemes – almost as many as usual characters…

When you move from the character level to the string level another huge amount of complexity mounts up. For instance, the web programming language PHP has at least 98 functions [62] just to perform standard operations over strings, and of course they are picky regarding special characters and only work over sequences of characters and not on any other sequences of "things".

Another practical problem with characters is that different characters are actually perceived as the same. For instance: "A", "a", "á", "À", "ã", "Ä" are just variations of the same perceived

character A, and you cannot universally agree on the order to use between these variants, creating a problem in sorting data. Also if a user performs a search by “A”, he expects to get results with all those variants.

A huge amount of works must be performed by programmers just to handle strings in a standard program and this is an accidental complexity. A string is a sequence of items, and many operations performed on strings would be really simplified if they were handled as sequences, instead of handling the inherent complex nature of characters.

Date / Time

We represent time (**moments** and **durations**) with multiple dimensions that are not multiple of each other: seconds (base 60), minutes (base 60), hours (base 12 or 24), day (base 28,29,30 or 31 relative to month or base 365 or 366 relative to year), week (7 days), month (base 12), year (base 10). We also have time zones so that the sun hours seem similar across locations, although they vary creatively across the map. We even have summer time and winter time, moving forward and backward our clocks according to local regulations. It’s so fun, we can’t possibly change that, or could we?

We have such a complex system by historical reasons – Sumerians created the sexagesimal system (base 60) probably because it was a number with many factors. Egyptians created the base 12 system allegedly because they counted finger joints in one hand instead of fingers in both hands. Twelve is also a number with many divisors (2,3,4,6). Also the day time and night time was divided in 12 parts (hours), and depending on the seasons the duration of hours were shorter or longer. So hours were not a fixed amount of time but a proportion of the day time and night time. This is why we still have AM and PM divisions, although we now refer to the mid-day and not the dawn.

The Gregorian calendar dates back to 1582 and is a mess with months that don’t match the year duration and are not regular in size. It’s not even a universal calendar, just a western calendar – many countries use a lunar based calendar. Other calendars, like the traditional Chinese calendar, also use the year as basis, but uses a different day for the year changing. Linus Trovals – the creator of Linux – choose Jan 1, 1970 00:00:00 UTC as its zero reference time, counting 86 400 seconds per day thereafter. This number is not actually exact as leap seconds have sometimes to be inserted in order to map to the actual earth rotation… [91]

During the French Revolution (1789) a new calendar [92] was approved and used for 12 years. It established fixed size months of 30 days, each composed of 3 weeks of 10 days. At the end of the year there were free 5 or 6 days for end of year celebrations. It also established that the day was divided into 10 hours, each hour divided into 100 decimal minutes, and each minute into 100 decimal seconds.

Currently, our “second” is the base unit of time in the International System of Units (SI) defined as [93]: “the duration of 9 192 631 770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the cesium 133 atom”.

We can realize the amount of unnecessary complexity added by our current time representational system when we try to explain it to a child… But why do we keep these messy old systems? We think that the only reason is a fiddler on the roof playing “tradition” [94]. For how many more years or centuries will the current calendar last?

Proposed Solution for Basic Types

The ancient philosopher Aristotle divided issues between essential and accidental complexities. The essential are the things that are intrinsic complex in the nature, while the accidental are those that arise from the options we make how to handle them and that are preventable if we use alternatives.

In the previous sections we addressed ways of representing basic data types that suffer from accidental problems due to the way we choose to represent them.

Our proposed solution as basic data types are just **Symbols**, **Numbers**, **Moments** and **Durations**.

Symbols

There should be an open table with symbols that can grow over time. There can be several symbols mapped to the same letter solving the multiple representation problem. Although storage of the actual messages sent by the user should be kept with the exact symbols, search should be performed at the token/letter level, abstracting from that level of detail. For symbols that cannot be mapped to letters, they will still be mapped as symbols and can be combined in sequences with letters using a Fly Weight pattern [95] as described below in the section 7.1.5 for structured data pots.

Symbols should be encoded with different bit size as they are used with different frequencies. The frequency of usage of each letter is sufficient information to determine the language is being used, without checking a single word in the dictionary. This is now a common feature in text editors.

Although we usually use bytes as the main size reference for data, raw data can be defined in any bit length and even in variable bit length in compressed files, using Shannon's Information Theory [96]. Therefore, raw data, unless has a known format that is able to be parsed, should be treated as a single block of data with a certain size in bytes.

A compelling example of the utility of using the symbols/tokens approach whereby presented comes from the bioinformatics that handles with huge sequences of DNA [97]. Although genes are expressed with only four letters T, C, A, G (2 bits), creating predictable pairs in the sequence, the actual way these data is currently expressed is in "codons" of 3 letters (6 bits) that are actually interpreted in 21 functional groups (less than 5 bits), including "codons" of sequence initialization and termination. Some variations within the same group can be functionally interpreted as the "same thing", because they are common variations that do not have functional differences in the encoding. Some groups have just one possible sequence, while others groups have 4 or more alternative "codons". Searching for similar sequences is not an exact match, but a match with functional equivalence using the groups of "codons". The ability of grouping different symbols as the same abstractions allows handling the complex nature of DNA is a much simpler ways, and therefore opens the perspective of better performance for future algorithms on this new and promising field of research.

Numbers

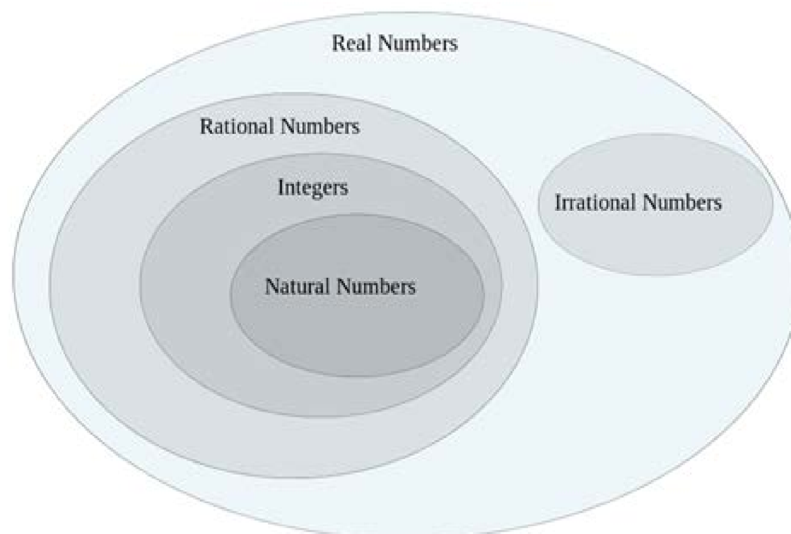


Figure 19 - Usual sets of numbers in mathematics

Numbers in mathematics are classified as Natural (N), Whole (N0), Integer (Z), Rational (Q) or Irrational and Real (R), Complex (C) and some other less common classifications. This classification is significantly different from the one described in the section above. A number can be represented in different bases; the most common ones are base 2, 8, 10 and 16. It can also be represented with different conventions like the Roman numerals, fractions or the scientific notation.

Independently of the type, base or form of representation of a number each form maps to a unique concept. For example: "2100" and "2.1E3" are the exact same concept. No programming language currently known by the author is able to parse a number like "1,322,145.12343(37)E-5" as part of its standard libraries.

There is also a common conceptual bias that makes us think that big numbers are equally likely in nature (or in human society). Actually they are not. Benford's law [98][99] discovered in 1938, states that the distribution of the first digit of real numeric data is not uniform. If the distribution was uniform for all 9 digits (numbers don't start with zero) the probability of each digit should be around 11,11(1)%. Actually, many data sets have shown that numbers starting with 1 are about 30% of all numbers, starting with 2 are about 18%, starting with 3 are about 13%, and all others are below 10% in their natural order ending at 9 with around 5%.

There are many real data to prove this law, and there is even a site [100] to check it in several sets of data. If data does not follow this pattern it might not be real data and this is commonly used in security checks as they might have been randomly generated or there might be any other strange factors involved, especially if the pattern changes overtime. As the mathematician Robert Coveyou said "The generation of random numbers is too important to be left to chance."

This likelihood distribution remind us that using a equally likely way to represent numbers, like the sequence of decimal digits might not be the most compact representation according to the Shannon's Information Theory [96].

The same number concept can be represented in different ways, but the representation used has an impact on the performance and the error handling of the system. For example: if you divide the number 7 by 3, and then multiply the result by 3 it is not guaranteed that the final result is 7, but a number nearby. In general, floating point's representation on computers cannot guarantee correct answers to equality tests.

Although numbers could have infinite precision there are patterns of usage, namely linear spaces and portions. Linear spaces are sequence of numbers with upper and lower bounds and equal intervals. They are usually aligned with decimal representation therefore can be represented with a certain number of decimal places. Portions on the other hand are created by dividing a range by a certain number. The issue is that dividing by any prime number not used in the factorization of the numerator results in an infinite precision number. This is an issue that cannot be solved with the most common way of representing numbers that is a sequence of arbitrary size of decimal digits.

There are several ways to represent integer numbers. Computers use a fixed size format to represent numbers, which lead to a huge amount of underflow and overflow problems.

In my opinion, programming languages should not represent number in fixed size formats, but in an upper level conceptual level. Translating upper level concepts into numerical operations should be a task left to the inner working of the system.

This author proposes that number should be represented as concepts. For each number concept there are dual complementary representations, namely, the usual sequence of decimal digits that is adequate for sums and subtractions, but also a representation based on product of primes (with an exponent) that is adequate for products and divisions.

According to the Fundamental Theorem of Calculus, every natural number can be expressed by a unique product of prime numbers (apart for order of terms). There are infinite number of

primes, but bigger primes are less likely to appear in nature. As integer numbers get larger only 2% of them are primes.

$$N_i = {P_{i1}}^{e1}\ x\ {P_{i2}}^{e2}\ x \ldots x\ {P_{in}}^{en}$$

Each prime only occurs once in the expression and the exponent is used to express the multiplicity of each prime number. Unused primes is like having zero as exponent, therefore counting as a neural element in the multiplication because any number raised to zero is equal to one. This kind of representation is really useful for multiplications and divisions because of exponent properties in multiplication: when multiplying numbers, some primes are shared and others are not. The unshared primes are added to the result with the same exponent as if we where using a union in a set operation. The shared primes are represented adding the exponents.

The same happens in division, but subtracting the exponents. All rational numbers (Q) can be represented by this expression with some of the primes with negative exponents. Primes with exponent zero should be removed from the result set. Rational numbers are enough to represent all measurable quantities in nature.

There are other special case numbers that should be added to the universe of numbers like Zero (0), Infinite (oo), plus Infinite (+oo), minus infinite (-oo) and NotANumber (NaN). These are special case number concepts that may occur as a result of certain operations, for instance, dividing $\frac{0}{0}$ you get NaN. With these add ons, numbers can be treated as a Universal Algebra without ever causing exceptions.

If you add a Boolean to signal positive or negative numbers you get a way of representing all Integer numbers. The minus sign has no effect on the zero number. This is useful either for this prime product representation as well as with sequence of decimal digits. Whole numbers (N0) are all naturals plus zero.

There are many irrational numbers but only a few of them are actually useful like PI (π) and Euler number (e). These special numbers could also be expressed as number concepts.

Finally, in order to be able to represent almost all real numbers the missing structure is to allow the exponent of the primes to become a fraction for each of them. With this change we could express all root based expressions, because $\sqrt[3]{x} = x^{\frac{1}{3}}$.

The representation of Complex (C) numbers could be obtained by an additional number concept to abstract $i = \sqrt{-1}$ and use two dimensions to register real and imaginary numbers.

All calculus expressions can be obtained from the combination of the previous concepts with custom operations like exponent, logarithm, multiplication, division, addition and subtraction, series, integral.

The ability to express numbers as concepts with multiple representations allows the use of strategy pattern to perform operations – that is – choosing different methods to solve a problem because different ways of representation can lead to easier or more difficult solutions.

Another argument in favor of representing each number as a concept is that when you face a number like 2014, you immediately perceived it as a year – especially now that is the current year. Also if due to frequency of usage you buy a certain product with a distinct price and later on you see that same unlikely price on another price tag your brain will immediacy connect the two facts. By considering a number as a mere value, instead of a concept, you cannot get the same immediate association.

Finally, this author believes that each number used in a Information System should always have an associated error (implicit error). In some cases the error might be zero, but typically, in data obtained from measurements the default error it is half the size of the last significant digit. For example, if we measure a distance with 12,27cm the default associated error would be 0,005cm. The importance of this value is that for this is because computers were built to make hundreds, thousands or millions of calculus over data very fast. We don't notice that amount of

calculations a computer is performing, but implicit errors could be mounting as operations are performed and that should be a relevant information presented to the user, when required.

Moments and Durations

Precise timing is a human construction that is not at all accurate. Mishandling of the time issue is a major source of complexity that needs to be handled in a better way by information systems. Time is a quantitative, interval type of data without an absolute zero, even so we insist on representing time using a fixed zero, which is the cause of a lot of problems.

When you see the following three dates: March 1, 2013; Jan 1, 2014; December 1, 2014 you immediately see a pattern in these dates (first day of the month) that is impossible to catch if you identify these dates by the equivalent numbers: 41334, 41640 and 41974. The same could happen in any other of the usual dimensions for reference time.

To solve this problem we should always store time as a reference, in several time dimensions, or in seconds (or both) in reference to another date that depends on the context. Using a universal reference is useful for some rare calculations but not for normal human usage. Humans use time as relative measurement and not as an absolute measurement.

When we consider the scenario of a network of computers, each with different time clocks the scenario gets even worse as clocks are not that precise – even the ones used on our computers. Synchronization algorithms can do a good job, but they only work by approximation.

When we refer to time we can be referring to two situations:

- a moment with a predefined precision (day, minute, second,…), eventually in relation to another reference time (day, year, …)
- a duration that is a moment plus a rational number in one or more time dimension (second, minute, hour, day, week, month, etc) to represent an interval

Therefore we choose to represent moments in time and durations as multi-dimension entities. The dimensions could be relative to the second (the unit in the international system), the day, the week, the moon cycle duration, the month, the year, etc. This list could be expanded in the future to accommodate new references that become useful.

A moment in time (with certain precision) is also a concept that can be associated to many other things – special national or world events, events in the life of persons or organizations, etc.

Although there is a mathematical need of setting intervals with precise definitions (open or closed intervals) and clear boundaries, in human conversations time loses precision as we refer to distant periods from current time. When a person's says a week in the last month, it could actually be 8 days or even just 4 workdays. When a person says last week, the time period is better defined. Allowing this kind of ambiguity could help in getting results in a search if the stricter criteria is not getting the intended results. This flexibility produces interesting results in information visualization field of study.

Therefore, following axioms 1,3,4 and 7, as well as the reusable time ontology [101] and the Date-Time Vocabulary defined by OMG [102] as references we shall allow setting flexible dimensions, and when possible, establish conversion action between them to allow each moment concept to be defined in the most complete possible way.

7.1.2 Concepts, Objects and Types

Due to the success of Object Oriented programming paradigm the idea of object, class and type have become very rigid in most programmers head. An object is an instance of a class. A class sets a type with the corresponding constraints to the values it can hold. Classes can inherit from other classes expanding its definition and allowing polymorphism for its methods. The initial flexibility that existed in C++ allowing the inheritance from multiple classes was lost in most of the more recent languages and inheritance become a unary relationship. It's true that having the

same name for properties and methods introduced an additional level of complexity, but it is a useful feature.

The need of an object to be an instance of several classes exists very frequently and poses a problem in most of OO programming languages. To overcome those difficulties these languages introduced the notion of “mixins” for creating objects of several types simultaneously or the more fragile solution of “interfaces” (like in Java) where objects satisfy a certain pattern of methods API, but do not actually share properties of an inner state as object/classes do.

Typed constraints get even tighter when we consider patterns like Type Square [103] that constrain not only the entity to an entity type, but also their properties to property types that are also part of the same entity type. This type of restrict construction creates unnecessary rigidity that contradict the general goal proposed by Adaptive Object Models.

Another unnecessary constraint was introduced with great acceptance by Model Driven Applications [104] is the definition of a finite set of meta-model levels: M0, M1, M2 and M3. At the meta-level M0 are the instances with the actual data records. At the meta-level M1 exists the class (or table) definitions also with their property definitions. At meta-level M2 are the instances of the MOF constructs, i.e. the “Meta Object Facility” (MOF). And finally at level M3 the MOF itself, which is the “set of constructs used to define the meta-models”.

This author adheres to the much more agile and refreshing idea, although less known and accepted, proposed by XModeler (also called XMF – Executable Meta-Modeling) [105] that is a model driven development platform where classes can be at any meta-level. A class can be abstract or not – that is, be able to be instantiated, but at the same time its attributes and methods can be defined with an arbitrary ability of being instantiated at the level the programmer feels more fit. This allows an arbitrary number of meta-levels and a much more flexible architecture that promotes reuse of code although more conceptually difficult to grasp due to being so different from usual.

According to the meaning triangle [14], objects exist on the real world, and concepts live on the conceptual world. When the “real world” is a software program these differences become more difficult to grasp… But in the real world the differences are clear. For example: a specific horse can be simultaneously an instance of the class of mammals, an instance of the class of means of transportations and an instance of the class of athlete in a horse race competition. There is no natural way of putting “mammal”, “means of transportation” and “athlete” in any reasonable and commonly accepted taxonomy of classes.

In this work we allow objects to belong (be instances) to many classes at the same time. Actually a class is just like an object, the only difference being that some properties or methods are not yet instantiated. Even so, non-abstract classes can be instantiated with the limitation of some of its elements being unusable.

Objects (or classes) can have whatever properties or methods they aim with two constraints: having a method that produces a hash (a summary needed for indexing purposes) and the method that allow the object to be transformed into a sequence of text, in order to be able to be transferred over the network to other executing environment.

7.1.3 Identifiers

Having a unique global identifier for each object in a network environment poses several difficult problems.

- The problem of size of the identifier itself – how much is big enough? Having 64 bits allows addressing 1.6E+19 elements which seems like a pretty big number (16 Exbibytes), but certainly not big enough to identify all the atoms that exist in the universe, that is estimated to be between 10^{78} to 10^{82}.

- Having different stakeholders in the Information System and using axioms 1 and 3 as reference, we must allow for multiple perspectives over the same objects, and therefore allow them to be mapped as different objects be different stakeholders.
- The problem of guarantee of identifier uniqueness – that is, not allowing several execution environments to use the same identifier. To solve this problem either we have a unique key for each working environment and combine it with a time identifier or a counter, or have same kind of centralized authority that manages pools of identifiers.

Choosing a solution that depends on the existence of certain features of each execution environment creates a constraint that might constraint future uses of equipment. In the past there have been attempts by Intel to establish a unique ID Number for each CPU, with large controversy, and Network Cards actually use a unique identifier, but a network card does not uniquely identify a unique user. Also the use of impure identifiers, that is, identifiers that give some hint of the originator or creation time might pose questions of security that add additional concerns.

Ubiquitous computing promises that we will have several computers as wearable's, as well as access to many other devices that we may use occasionally [106]. Therefore, as less hardware constraints we demand from devices, better could be the adoption.

On the other hand, a naive centralized solution for getting identifiers would certainly become a bottleneck. Therefore we adopt the so common approach in Information Systems of adding several layers of indirection.

This work shall use a system of two identifiers: Category and Index. Category is centrally managed by the system giving to each execution environment a pool of available addresses that could be freely used by that execution environment at will. When half those addresses are used, the execution environment should ask for more and be provided in a timely manner without interrupting normal operations. For each category there is a shared description that manages the objects of that category, namely the amount of instances used.

The Indexing structure is based on blocks of 8 bits (byte) aligned in blocks of 64 bits. The two higher order bits in each block of 8 are reserved with the following meaning.

- If the higher order bit is 1, then this block is the beginning of an identifier.
- If the second higher order bit is 1, then this block is the end of an identifier.

In the image below we can see that the first identifier uses 2 bytes with 12 bits in total, and the second identifier uses only 1 byte with 6 bits, as the block starts and ends in itself.

The full identifier always uses multiples of 64 bits, that is 8 blocks with the structure below. The blocks needed to the identifier for the category is always align to the left (higher order blocks).

The identifier for the index has its 8 bits blocks always align to the right (lower order blocks).

All other blocks required to fill in multiples of 64 bits are marked with 0 in the content and in the 2 control bits.

With this identifier scheme it is possible to have infinite identifiers for categories and indexes, but at the same time have an economic representation that "only" loose 25% of the available bits in the 64 bit.

Figure 20 - Indexing structure for infinite size identifiers

When more than one execution environment starts using the same category, with due approval by its owner, pools of indexes are made available to that environment and that category, in order to allow new instances to be created in that environment with unique identifiers.

In general, as each person uses a limited amount of devices, the number of possible sources for requests for each category is limited by nature.

The objects addressed hereby described are not actually a block of data but again a reference to the shared memory. It is another level of indirection to allow the proper memory management of the system.

This system for creating identifiers allows the creation of pure identifiers that can be shared on a network with the ability to grow over time, but with minimal demand from the centralized system, therefore avoiding the creation of bottlenecks.

7.1.4 Basic Data Pot

Data Pot is an essential building block of this work. A Data Pot is version controlled closure that stores values, its history of stored values, references to the other elements that used it, its value constraints and its interface with the outside world.

Data Box is a complex element, but as Einstein said: "Things should be as simple as possible, but not simpler."

A Data Pot is a black box in the sense that only itself controls the way it is internally organized. A Data Pot uses blocks of memory proportional to 4Kbytes. The reason for this it to map each structure with the current minimal size used in the computer L1 caches. Overtime this definition may change. A Data Pot can use as many memory block as it requires and the references to the remaining memory blocks are kept internally.

Each Data Pot has a version number (major and minor) as it is supposed to evolve over time with bounded effects over the remaining of the system as prescribed by Normalized Systems.

Value, Set of Values and Weighted Values

A Data Pot is used to store data of any kind. It can be a literal like a concept, a number, a letter, a weekday, etc. But it can also much more complex data in structured, semi-structured or unstructured form.

- **Structured data** is stored in repositories like databases where information is organized and relational form, even if its semantic might not be fully presented in a structured way.
- **Semi-structured data** is data that is in digital form, maybe in a file in a logical folder, even if in a format that is not easy to handle like text documents, image scans, movies, audio files or other known file type.
- **Unstructured data** is all other data that is in an unknown file format or even not accessible from a computer in the system, that is, unreachable data in computers, in paper (even if in an organized archive) or in people minds.

Structured and Semi-structured data is kept with Structural Data Pots described in section 7.1.5.

A Data Pot can also be a **variable** as it can point to other object. If the Data Pot is a variable then it will store the identifier of the category and the index (variable size bits in multiples of 64 bits).

Unlike the variables on usual programming languages, in this work variables can store many values at the same time. The several values are stored in a set of infinite size. This might seem a strange concept, as it's so different from usual, but it's a foundational concept and really important one.

Literal Data Pots are created with the immutable information they will hold forever. Variable Data Pots are created empty. A value is set into the Data Pot through the Set method passing a

new value (identifier of category and index). If a second Set method is called a second value will be added to the set, unless it's an already existing value and in that case the weight value is incremented. More details on weighted values later.

Data pots keep their history and are able to move back in time to answer to questions about its past using Memento pattern. [107]

In order to have a variable to work with the usual semantics of variables in programming languages you have to order the Data Pot to Forget (or Unset – syntactic sugar for the same semantic) a specific value or to Forget All and return to the empty set format.

This construction has several benefits: allows handling ambiguity, different opinions and inconsistencies of the way we represent the world and our current model. This is consistent with axioms 3,5,6 and 9.

This way to handle ambiguity is also useful in a multiprocessor network, where messages arrive asynchronously and we cannot assure they arrive in the proper order. This way of working prevents the problems of races. Also, Set and Forget messages can be used with idenpondent semantics, that is, repeating the same message produces the same result if you are considering the use of only storing one result at a time. By a small period of time the variable might have no value at all, and in other small period of time there might be two values stored, but that is not a problem as we assume that there is no global God keeping track of time.

For instance, if you ask a group of friends what is the color of this car? You can get 10 answer, with 5 saying "blue", 3 saying "dark blue", 1 "metalic blue" and 1 "deep ocean blue". And it would be just fine!

Values can be of nominal, ordinal, interval or ratio type. The DataPot does not make any distinction on the type of accepted values, except if instructed to do so with Constraints.

All data set are stored with provenance meta-data information – that is, from whom (Action Pot) was these information sent and when did it arrived and was processed.

The Set method allows using a sequence (Structured Data Pot) of values to do multiple Set operations at once.

If the Data Pot holds several values, when someone calls the Get method the several results will be returned in a sequence (Structured Data Pot).

Each value also has as an implicit or explicit associated error that is used to calculate error propagation in mathematical operations.

Sometimes values also link to a unit of measure, although not mandatory. Unit of measurement allows the automatic conversion to any other comparable measure, which can have benefits in terms of search, comparability and the way data is presented to the user. As a general rule, data will be stored with the values, format and units that came in the original message, although automatic processing is applied to perceive it as the proper number.

Getting back to the issue of weights in the possible values of variables, we must argue in favor of this representation, as concepts are stored uniquely in the system through Literal Data Pots, therefore there is no need to add repetition and equality is guaranteed. However it's not possible to have negative values on weights. The weight is a number, therefore, what is stored is a pointer to a Literal Data Pot with that number.

Name and Identifier

A Data Pot Variable can have a Name. The name is not an unique identifier, but can be useful to use in the context of a closure like a Action Pot.

The real unique identifier is automatically created when the Data Pot is created, and is a unique global identifier as described in the section 7.1.3 Identifiers. To get the identifier the method Get ID is available. Notice that in case of a Variable Data Pot, if the Get ID is called on the

variable the result would not be the same as if the same method is called on the Data Pot that results from the call to Get.

Constraints

A Data Pot can contain multiple constraints associated. Constraints can be described as Data Pots or as Action Pots.

In the case of Data Pots constraints can establish that only certain type of category can be hold by the Data Pot. For instance: only numbers. It can also establish a range of values, upper bound, lower bound or linear space of possible values. Among the linear spaces are the interval repeated values that can be produced by functions like *arcsin*.

Constraints can be defined as Conjunction or Disjunction of a sequence of constraints, allowing to define a very complex system of possible values. Data Pot constraints can be used in combination or in alternative to Values.

Constraints established by Action Pots are the black box scenario where the evaluation is performed externally. The Constrain Data Pot keeps a reference to the Action Pot and calls it when a Set method is called. Action Pot Constrains can also be combined in Conjunction or Dijunction sequences.

The tactic of using Data Pot with constraints instead of values can be usefull for solving problems of Linear Programming as well as many Logical Inference problems, like in Prolog

The constraints can change over time, even during the execution of the problem and that can actually be part of the search for the solution of the problem by removing invalid or unlikely possible alternatives. For instance, solving the Sudoku game.

Finally, a possible additional usage of the Constraints and the Values in combination is the use of statistical simulation on wicket problems as a novel way of solving problems. Some problems are hard to solve in iterative ways until getting to a solution, but an alternative way of simulating a solution is to generate random values to all the variables and check if all of them satisfy the constraints.

- If yes, then Set that solution as a tupple of possible answer to the problem, and also Set in the variable this possible solution with the purpose of getting a histogram for each variable of range of possible solutions.
- If no, then ignore the solution all together using the Tabula Rasa rule, that is, do not attempt to adjust the values of the random values in order to get a viable solution, as you would be adding bias.

The computer can simulate millions of combinations of random numbers very quickly, and even if it takes several million combinations in order to produce a viable result, many could be found in a relative short period of time.

For many wicket problems that are hard to solve iteratively this can be an alternative solving method, as well as a starting point to get a feasibility range of possible solutions and start from there to try to get an local optimal solution.

Getter and Setter Notification Subscription

Action Pots can subscribe to the Get and/or Set methods in order to be notified of calls to these methods, receiving the same input message received by the object. These act asynchronously and as a complementary call, not as an alternative to the Get/Set methods.

Subscribing to these notification systems is subject to approval by the Data Pot owner.

Managing References

Data Pots keeps references to the other Data Pot and Action Pots that are referencing it. An exception to this rule might occur in case of frequently used Data Pot Literals that have so many references that doesn't make sense to keep a record of them for operational reasons.

This information is relevant for the memory management system and Data Pot Variables might be created within closures of Action Pots or Structured Data Pots and later on loose those references without being deleted. These references are alternative to mark and sweep algorithms of garbage collection.

Having these References is also useful to link unrelated things that might actualy give an insightful information to the user. This may occur due to singularities, but also through the use of data analysis techniques based on levels of support (number of occurrences of some "situation" on a general scenario) and confidence (number of occurrences in "situation" that satisfy a certain condition).

Equality functions

Being equal is a complex decision as it depends on the context. "It takes talent to find differences in similar things and similarities in different things."

It should be possible to assign many equality functions with different criteria of determining if something is equal or not. The several equality functions could be called specifically or in a strategy pattern to get a majority of votes. For example, in the domain of colors, two tones of orange might be perceived as equal in some contexts, and different in others. Therefore different equality functions have to be used.

7.1.5 Structured Data Pot

Structured Data Pot's are use to store all typical data structures that are referenced in the literature review like sequences (lists, rings), associative maps, trees, graphs, etc. One of the major contributions of this work is the discovery of a single compact way of describing all those structures with a Universal Data Structure.

7.1.6 Universal Data Structure

All data structures all build based on "boxes and arrows", that is, nodes and edges between them. Nodes can by divided in 3 parts: the ones that are at the beginning, at the middle and at the end. Sometimes there are just middle elements, like in a ring, but typically there are also start and end items.

To describe the structure we must define cardinalities (min and max) for the nodes, and cardinalities for their connections to the previous and to the next elements. Cardinalities can be zero in the case of absence of those kinds of structures, but can also be void to allow infinite number of elements.

With these simple constraints we can represent all data structures that we know. In the next page diagram we can find a structural representation of this Universal Data Structure, giving to each element its properties in terms of name, type, default value, instantiation value and instantiation information.

This representation uses the flexible instantiation structure described in section 7.1.2. Therefore, each structural element has the information at what level he is in, and each property has the information at which level it is expected to be instantiated with a value, otherwise using the default value.

It is possible to store content either on the node or on the edges. Content stored if a Data Pot with any content. It can even be another Structural Data Pot. It is also possible to establish content type constraints, forcing all content in nodes of being of the same type.

Edges can sometime the constraint of not allowing the repetition of the same node after it previously occurred previously (no back links) as is the case in trees.

Also edges can have the semantic information if the connected parts share parts, have essential parts and if there is an existential dependency, leading to the rich set of structural relations between elements described in section 4.2.3.

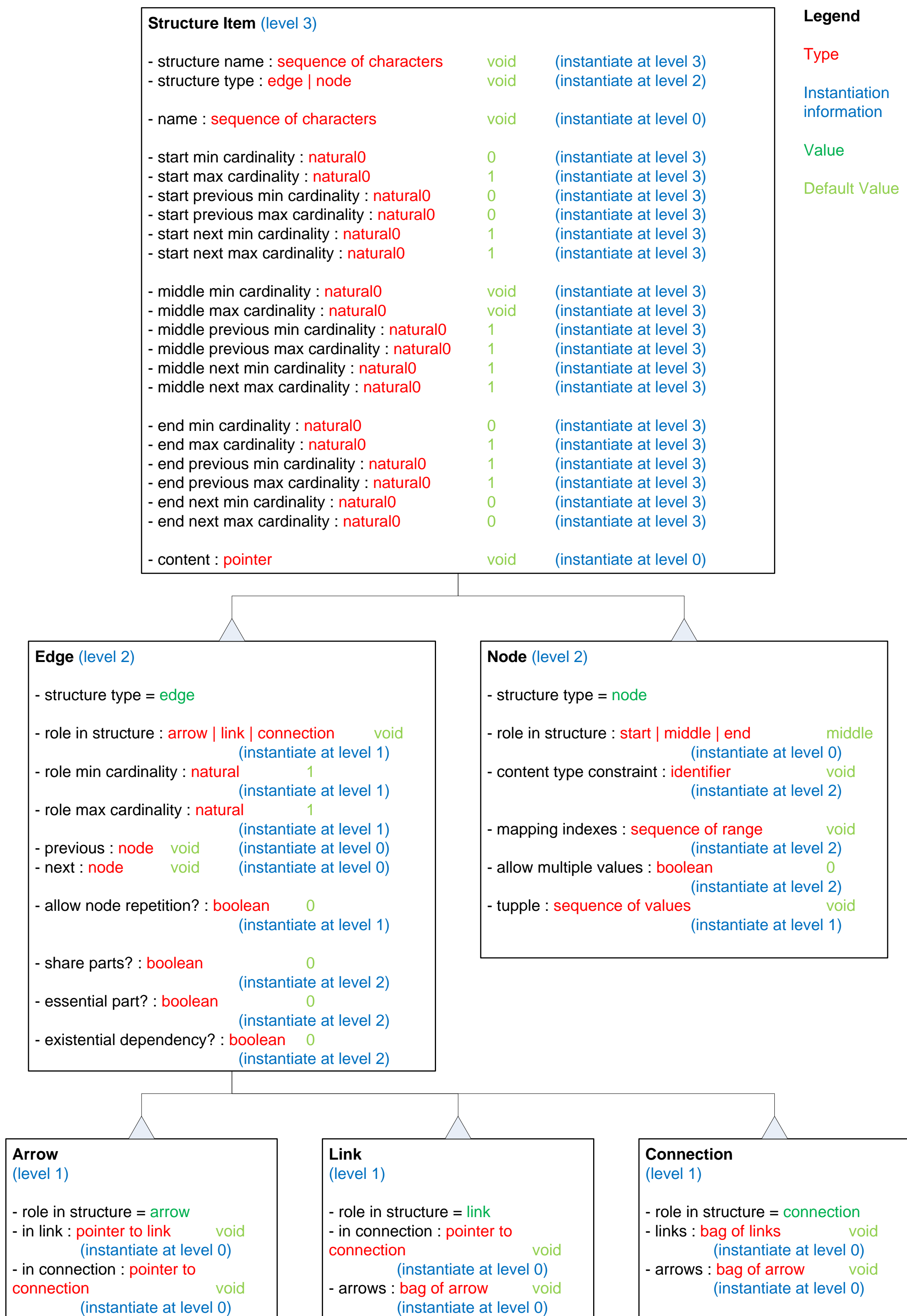
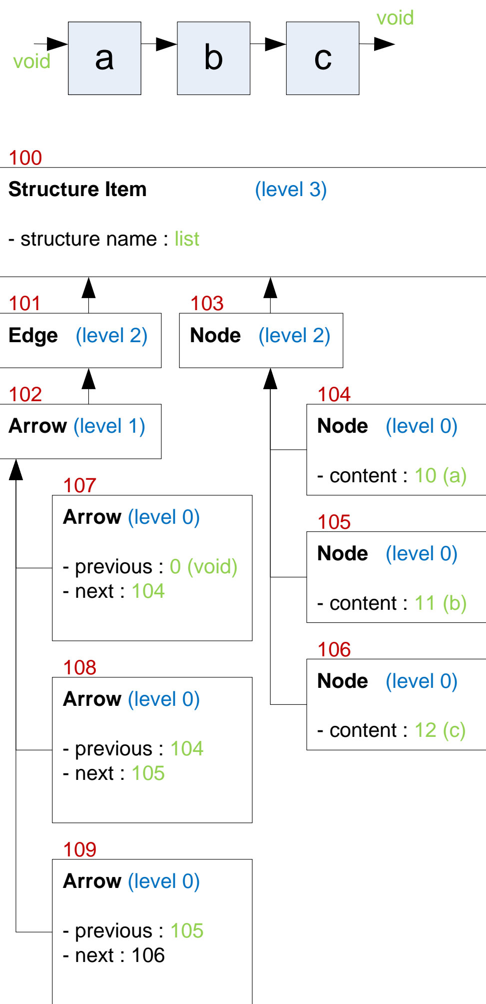


Figure 21 - Construction to hold Universal Data Structures



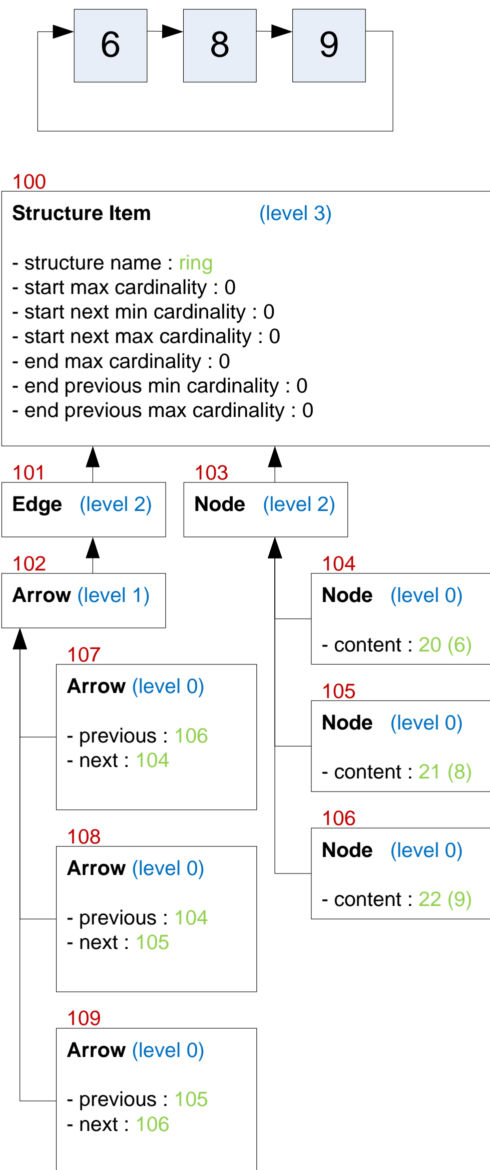


Figure 22 and 23 – Instantiation of simple list and ring structures

The instantiations presented in the above figure show how these data would be presented using the Universal Data Structure. Notice that with the default values for the cardinality fields you would get a sequence. For the ring data structure, the start and end types of structural elements would be set to cardinality zero.

In the next page we present a more complex structure using this construction: a binary tree. It is also possible to create a B-tree increasing the max cardinality and using the tupple and setting to true the property "allow multiple values". Notice that this property allows storing several Data Pots in the data structure (in tuples), which is not the same thing as allowing several values within the same Data Pot, already mentioned.

The property mapping indexes allows associating two sets of data, allowing the creation of associative arrays. In this case, the property "allow multiple values" becomes an indicator to whether collisions will be allowed or not in this data structure. Indexing structures will be presented in the section 7.1.6.

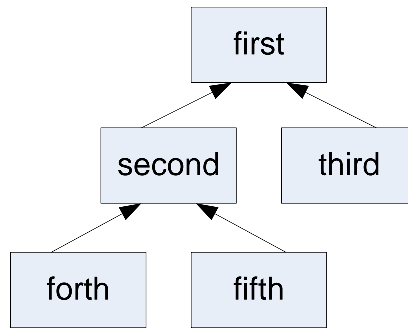


Figure 24 - Binary tree representation

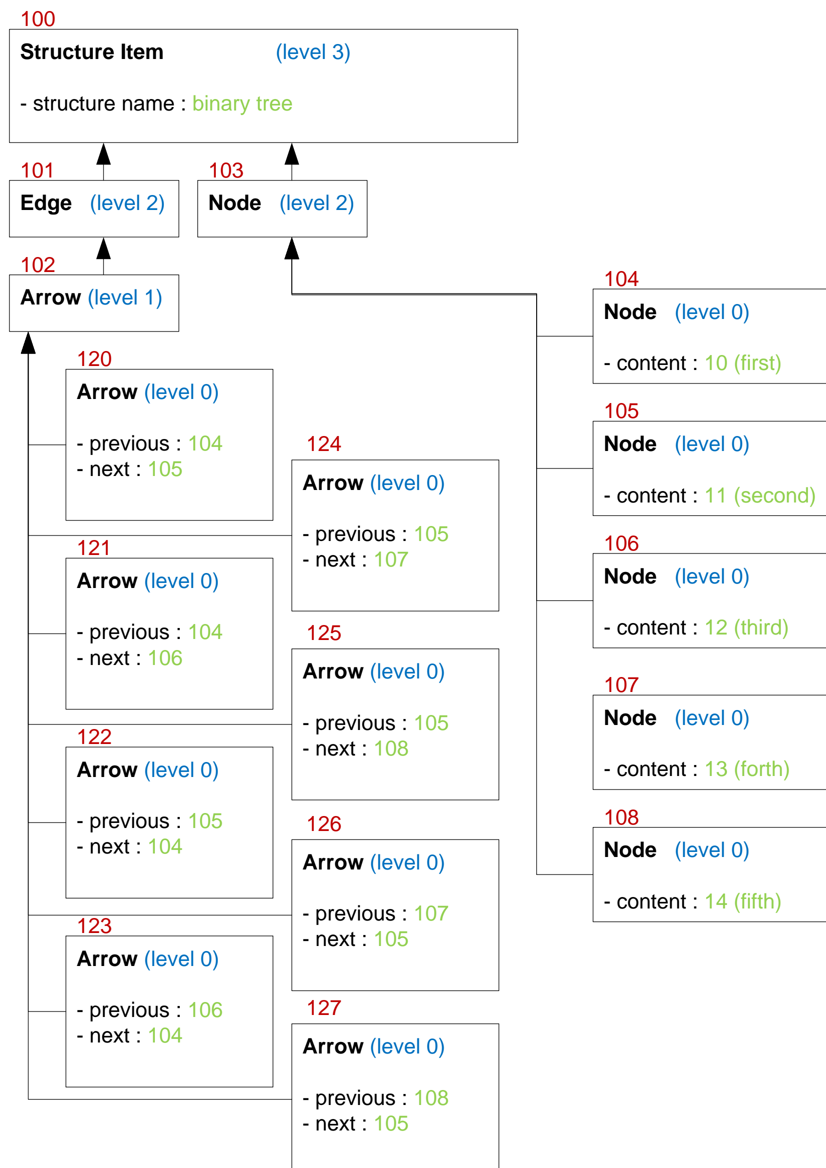


Figure 25 – Instantiation of a binary tree structure

7.1.7 Shared Data Pots

Sometimes a Variable Data Pot or a Structural Data Pot needs to exist in more than one execution environment at the same time, for instance when two users require access to it in their mobile devices.

In this scenario we get a scenario where both instances (called Elements in the diagram below) need to communicate with each other and with the central data repository in order to allow any of the users to Read (Get) and Write (Set or Forget) data without data inconsistencies.

Multicore-processors already face a similar problem as caches L1 and L2 and exclusive for each processor and only cache L3 is shared. Therefore the same block of memory can be loaded into the caches of more than one processor so that all of them are able to read and write without inconsistencies.

In order to handle this situation several cache state transition diagrams were created. The first one called MSI (Modified, Shared, Invalid), that was later enhanced to MESI (Modified, Exclusive but unmodified, Shared, Invalid) by adding an exclusive state when only one instance has a copy of the value (besides the central repository). The chosen solution was the MOESI (Modified exclusive, Owner, Exclusive but unmodified, Shared, Invalid) that introduces the Owner state that gives to a certain Element special preference, optimizing the process as long as others do not intent to write on it. This seems to be a reasonable solution as there is probably a dominant interface being used by any user at a specific time.

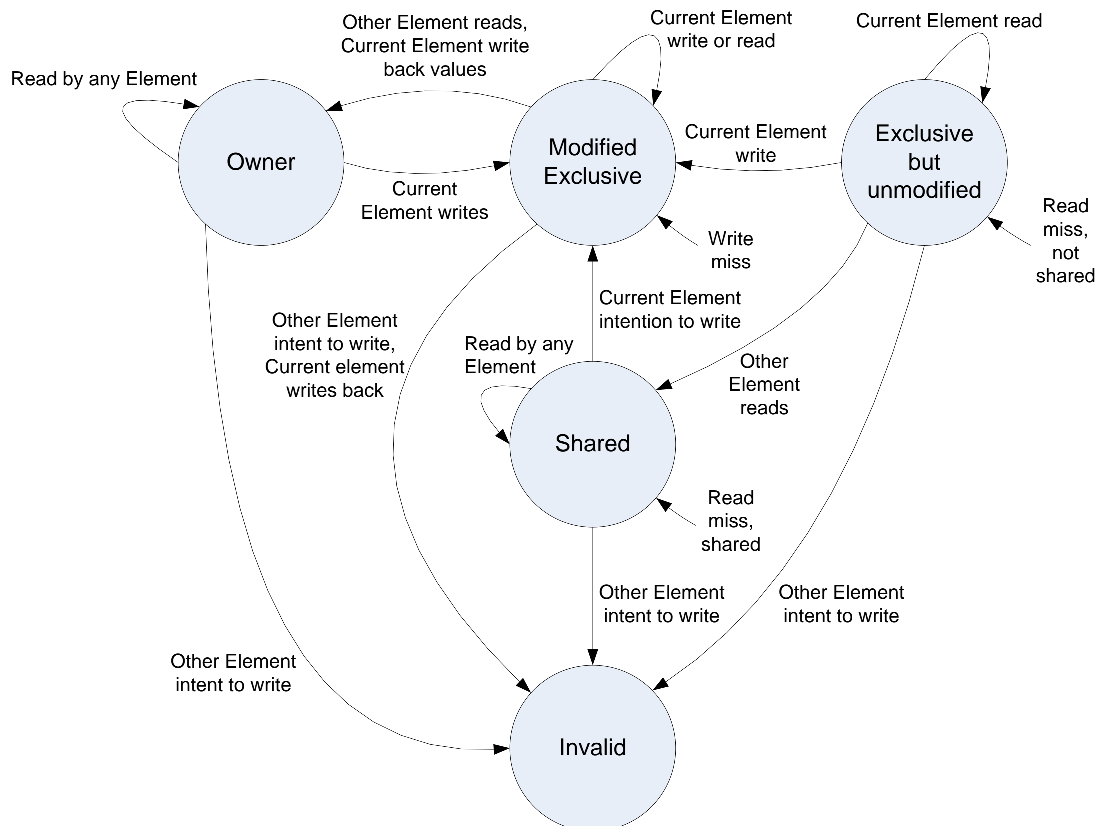


Figure 26 - Adapted MOESI shared memory state transition diagram (Used in AMD Opteron)

Intel Core i7 also implements a different state transition diagram to solve the same problem, called MESIF, where F stands for Forward, but this solution is harder to implement for the problem scenario due to requirements of direct communication between environments that could be difficult to implement in future devices.

7.1.8 Visual Programming

By abstracting Structured Data Pot in the same data holding mechanism like much simpler data, there is an opportunity to develop generic visualization techniques that use the Abstract Factory pattern. This allows make each Data Pot visually appealing and at the same time have the graceful degradation ability, that is have simpler, textual, representations, that could be more convenient when there are constraints in the devices being used.

Order Functions

Order functions are an essential feature to establish order relationships between elements. Together with Equality functions they form the basis to extent the applicability [108] of Universal Algebra to a vast field of applications.

There might be many order functions over the same set of data. Ordering based of each field of a table is an example. But even within the same field it is possible to establish order relationships.

For each full order function it is possible to create ordered indexes. Full order functions are the ones that are able to establish order relations between any two elements of the set. Index Data Pot will be presented in the next section.

The most common result of a order function is to return -1,0 or 1 stating respectively that the first element is before, equal to or after the second element. Other order functions that have a notion of the range of values in the domain, can return more informed results by returning rational numbers between [-1,0) and (0,1] to better inform the caller and therefore get better hints regarding the distance between items in the ordering. This could lead to better ordering algorithms.

A final topic about order functions is the possibility of partial order functions, that is, functions that are able to establish order relationships between some elements of the domain, but not between others. For example, you can easily establish an order relations between several colors of the same hue (lighter green, darker green), but it will be difficult to establish the same order relation between green and red.

Each ordering function should be implemented as an Action Pot, described in section 7.1.10.

7.1.9 Index Data Pot

Indexes are essential structures in Information Systems. Ordered indexed allows searching in loglinear time (N*log N), no matter how large is the dataset in use.

In general terms an Index is an association of elements between elements in one domain and elements in another domain that typically are more complex that the first ones.

Typically on the indexing domain we have whole numbers that start in zero and can increase as much as needed. The indexed elements can typically be of any kind. However, it is possible to have sequence of characters as indexing elements, as it happens in PHP and JavaScript with associative arrays that can use strings.

In this architecture we go a step further and state that any Data Pot can be an indexing structure. This enables the very powerful situation of allowing indexes that handle ambiguity, that is using as indexing Data Pots that can be many possible values at the same time. This will be explored in future work.

As the most common function of indexes if allow fast searches, it might seem strange that we use Structured Data Pot as an indexing. However Indexing Data Pots are much more than that. They are a map between domains, following Map Theory. We will further explore the possibilities this architecture opens in future work.

7.1.10 Basic Action Pot

Action Pots are the mechanism of transforming Data Pots into other Data Pots. Every Action Pot takes zero, one or more Data Pots as input and produces zero, one or more Data Pots as output. They look like functions and functional programming.

Functional programming is probably the oldest programming paradigm. The major constraints of function based programming are:

- Functions return only one result and many functions need to return more than one result item. The results might not always be of the same type, so there is also the need for polymorphism in the results.
- Functions put on the same location the error reporting and the function result, creating unnecessary complexity of the analysis of the function call.
- Most languages allow multiple end points for their code (especially in error cases). That typically results in not freeing up used memory, and that can be a problem for languages that don't have a garbage collection system. Even for those that have, reducing the amount of trash is always the best option…
- Some languages added exceptions to separate the error generation and handling from the normal stream of code execution. These add additional complexity because most times the exceptions are not handled at the appropriate location and the errors cascade to unpredictable locations making errors harder to detect and fix.
- Functions receive their input through parameters. The typical way parameters are used is copy them to stack before function call and then back to internal data. These copies occur as a standard procedure, but are really only needed if recursion happens (either directly or indirectly), otherwise they wouldn't be needed.
- Functions inside objects typically have input polymorphism, optional parameters and default values for parameters. By forcing function calls to be called with parameters in a specific order additional effort is required to the programmers. The programming language should be as fault tolerant as possible and try to guess the appropriate associations only complaining when there are doubts (even so, by trial and error most of them could be figured out and clarifications suggested to the programmer).
- Some languages introduced the additional complexity of allowing parameters by reference, that is, after the function code ends, the reference values are propagated backward.
- Functions typically can access to external variables violating one the most basic principles of functional programming. This access can either be by value or by reference.
- Functions have no memory – they don't know anything about what they did before or any prediction of what they will do in the future.
- Functions assume that their code can be performed in a serial, synchronous, single time execution, without possibility canceling from the caller, with the current existing resources and hopefully without blocking. In reality is it very easy to create an infinite function that never ends. Sometimes it is very difficult to find the logical bugs that make that happen on very specific input cases.
- Functions assume, as mathematicians usually do, that there is only one solution for function call. There are cases where there are multiple valid results.
- Functions assume that we need the best result possible, when many times getting an intermediate result would be good enough to decide to wait for a better result or quit immediately.

- Functions assume that the code is executed sequentially, but actually instructions within a function should be considered a partially orderly set (POSET) because not all instructions directly influence others, especially in the case of calling other functions.
- Some programming languages, like Python, have generator functions. These functions yield a result instead of returning one. Everytime a yield instruction is found a result is returned, but the function keeps track of current position and the closure of all its variables. On the next call the function will resume from the previous saved state.

Functions have a typical structure that results from the previous points:

1. Copy parameters from stack to local variables
2. Check for parameter consistency and eventually return errors
3. Initialize variables and required workspace (eventually return errors if requirements are not available)
4. Perform local code and check for local errors like division by zero or overflows
5. Eventually call other function meanwhile and check for errors in the function call and possibly return to the previous point
6. At some point in code prepare result for output (again using stack)
7. Free used resources like memory (or call functions that do so)
8. Terminate, updating the reference value parameters

We believe that:

- All function calls should be made asynchronous, even in real time systems. For those, there should be scheduling and priorities to assure the timing is done, whenever possible.
- All elements that influence result in a function should be marked as parameters, including random number generators (generator functions).
- All input and output parameters (including errors) of a function should be recorded in a log, as well as call time, total execution time, and resources used (if possible). It might seem excessive at first look, but consider the benefits: control, backtrack, error propagation and cached results. If input parameters are clear and the function result only depend on them, then input could be hashed and indexed. Then, every time it is called again we could launch in parallel a log(N) search on the hash index, N being the number of different calls to that function with different parameter, and a normal execution. Whichever found first would return the result, but repeated calls with the same input would not be left unnoticed, and counter would be updated in order to generate a top usage. Each search on the hash index is also a function call, so, in turn it would be faster because it would be also cached.
- All functions should use spreadsheets for input and output parameters. There should be named cells and automatic conversion of formats as parallel as possible. Spreadsheets are excellent ways to have parallel code expressed and dependencies between those lines of code. Besides that it also allows to separate parameter validation from code execution in the majority of the cases. Results can be required in many different formats; the output spreadsheet can handle format conversions easily, also in parallel. All cells in the input/output spreadsheet could be marked as eager evaluation (as soon as possible), lazy evaluation (on request – as late as possible) or scheduled evaluation (start after N milliseconds of completion – to prevent other possible updates from time/space locality, or at specific time intervals or sometime before some already scheduled event – that will probably consume that data).

- All internal variables in a functional should be mapped into a spreadsheet (logically different from input and output spreadsheets). By doing so many logical parallelism is immediately expressed and it is easier to find the logical dependencies on the partial ordered set of instructions.
- All functions should be prepared to run on a limited time frame. At any time storing, we could store the internal variable spreadsheet (that should include the current line of code) and that should be sufficient to interrupt the execution and return later without any loss of data or context. Probably the execution periods should be fixed around 10ms to 100ms. On a 3 Ghz this would mean $3x10^7$ or $3x10^8$ computer cycles, which should be enough for most basic functions since accessing disk, that can take much longer times, should be isolated as specific functions because of usage of system calls. Users take almost 250ms to move their eyes and detect changes on the screen, unless there are preemptive changes, as described before. Studies from games and human-computer interaction state that the best feedback time (strongest association between action and outcome) for actions performed by the users is after 200ms to 400ms the start of the action or midway through the completion of the action. A user typically feels he has been interrupted if there is a delay in the interface longer that 1 second.
- The typical access times [109] are the following for the several types of resources:

Resource	**CPU cycles**	**Available Size**
Registers	1 cycle	256 – 8000 bytes
Cache	3 cycles	256 – 1Mb
Main memory	~100 cycles	32Mb – 4Gb
Disk	500 – 5000 cycles	4Gb – 1Tb

All functions should have code expressed as parallel as possible. Code itself should be expressed in spreadsheets as argued in the next chapter.

Dating back to assembly code and data were different in nature. Code was predefined, punched in cards, difficult to change. On the other hand, data was flexible, mutable and handled dynamically. Then came the higher level languages, but code still needed to be compiled, which was a big effort in the beginning, and its results were saved to disk as persistent memory, and only run afterwards.

With the advances in compilers, the areas of lexical analysis, parsing and semantic analysis are no longer a problem today [109] and can be handled pretty easily resulting in small code that is easy to handle. The areas that are still challenging are the optimization and the code generation parts. Times have changed! Code is now very small compared to the amount of data they handle because the era of Big Data has arrived.

This evolution in compiling techniques allows five important new approaches:

- Interpreted languages can now be almost as efficient as compiled languages, because loading data from disk takes almost the same time as compiling in memory - these allows compile to memory approach;
- Grammars and finite automata can be added as basic structural elements for programming languages, allowing more powerful means of expressing constrains over data, but at the same time, making easier to handle text in a more natural language format;
- Code optimizations can be performed in the long run, testing alternative optimizations that take longer to check for utility and therefore are typically not used in standard

compile time. Also, optimizations can be performed considering actual input data and not just all possible input data.

- Since data is now so much bigger than code, so big it is sometimes impossible to copy it in acceptable time, it could make more sense to make code portable and send to code to be executed near the data and send back the result instead of transferring the data.
- Code instructions can be considered as any other data and give rise to a new generation of software based on mutable code that, until now, was only slightly used in viruses with wrong purposes.

Actually code is not so different from data – they are stored and loaded exactly as any other data. Each assembly instruction [110] is identified by an upcode that uniquely identifies it. Depending on the instruction upcode, the processor will take several input parameters with predefine size (either internal registers or memory locations) and changes those elements, either memory or registers or flags. The instruction set manuals state in a clear way which elements are changed as output of the operation (in predictable ways – defined by the instruction itself), which elements are changed to predefined values, which elements are changed in unpredictable ways (that is, no value/meaning can be trusted as result in that element) and which elements are reserved for future evolutions of assembly language. This information is essential to optimization and code generation.

Most assembly instructions consider the data in memory or in registers as if they had no type associated, or, operations assume an implicit data type and the operations is always performed. If data is not of that type then it makes sense that it is a problem to be handled by the programmer…

If we consider coding instructions as any other kind of data, that is, code that can be edited inline, then a new generation of optimizations can come to live:

- Elimination of variables that hold parameters by replacing at proper locations in the code to be executed.
- Elimination of branches in upfront code as soon as we know the input variables.
- Creation of alternative code strategies depending on the input parameters.

All the powerful uses that current computers are able to provide us are all performed by just applying sequences of assembler instructions on a CPU. Each assembler instruction takes arguments from pre-established registers or flags and updates either registers, flags or memory locations after performing the specified operation. Currently there are over 250 assembler instructions – they have been increasing in number over the years, but not necessarily in their use, as most compilers are still able to fulfill their job with the older set of instructions. Instructions have also changed over the years as architectures changed from 32 to 64 bits, for example. For this reason, all assembler instructions should be mapped as external actions, following the recommendation of Normalized Systems to create a solid frontier to face the external execution environment that might change over time.

Another type of basic action is the Control Flow Action Pot that basically handles sequences of Action Pot instructions as well as Go To’s, Whiles, Repeat-Until, If, Case and similar semantics. They basically hold a context and perform a sequence of assembly operations over it.

These correspond to the usual way computer programs are seen today – a sequence of computer instructions that is to executed sequentially, apart from jumps and branches. This is actually an old way of thinking that is not consistent with the current model of multi-core processors and super-scalar processors. In these kinds of computers you can perform many operations on the same time, over several data. The real problem are data dependencies, that is – read after write instructions and write after write instructions (with the assumption that only one value can be stored on each variable).

Action Pots can be seen as a mesh of data dependencies, that can be latter on translated into several sequences of instructions depending on the number of cores that the execution environment has available. Today, that number can as high as 16, but in the near future it could be 100 or 1000.

In the future Action Pots will almost always they run in parallel environments. Therefore programming based on meshes of data dependences become critical because of Gene Amdahl's law of parallel computation states that: on a massive parallel execution, the amount of sequential code (data dependencies) is deterministic for the maximum expected improvement of the algorithm time of the algorithm.

We envision this compiling operation as a compile to memory operations that is an adaptation to the environment – that is, create the blocks of memory with the assembler instructions to be executed (depending on the current architecture) and then marking those blocks of memory as being blocks of code and execute them on the several cores. This is a different approach to parallel programming that the usual GPU models or the Map-Reduce approach as it is centered on the code generation and not on the data being handled. This will be explored in future work.

Action Pots can be seen as functions in usual programming languages. But Action Pots do much more than functions:

1. First of all they are able to handle the ambiguity that Data Pots can have with multiple values and constraints and produce Data Pots that hold the cartesian product of all possible solutions. This can be quite handy, as multiple possible combinations may be invalid, and therefore removed from the output solution, but also, as the weights in the inputs can give weighted solutions that give much more powerful insights on the results.
2. Action Pots can have inverse actions associated for actions pairs like sin - arcsin, exponential – logarithm, multiplication – division. Sometimes it migh be easier to solve the inverse problem, either because it has been called in the past or because of technical complexity. Typical functions ignore the existence of inverse functions.
3. Action Pots can produce several answers to the same problem. For example, $x^2=4$ or $|x|=1$, and produce two valid results. The mathematical notion of function forces to have only one result. Typically functions return only one result, even when the same result is used for reporting error or solution – which is a major source of problems in programs.
4. Functions are assumed to give the right perfect answer and be executed as a deductive process – a list of instructions executed in predefined order either be code or be inference mechanisms like in logic programming. Today, there are other ways to get answers for many other types of problems: using statistical mechanics (random solutions and tabula rasa rule); using simulation and analyzing histogram of possible solutions; using linear programming; using shooting methods (ode23, ode45), like the ones use to solve differential equations; using neural networks that can give different (best effort) solutions on each execution, etc.
5. Functions are handled synchronously. They take as much time as needed and then return the "perfect answer". If isn't possible to cancel a function call and we can't do anything else until we get that answer. Many times we don't need the perfect answer and an approximate answer might be enough to what we need. Action Pots act based on asynchronous timed responses that get incrementally better (in subsequent messages) as time passes it that is possible and if the requester didn't ask to cancel the execution. There are some functions that are infinite by nature, for instance – the most accurate value for π.
6. Depending on the programming language there is a strict semantic about the evaluation of parameters. The most common one is immediately evaluate parameters to its current value and use that value. Other approaches are, for example the spreadsheet, where you set a cellule A3 to hold the function formula A1+A2 and every time those values

change you update the value in A3. There are a few programming languages that allow this semantic, but it's a rare case. Action Pots can use Lazy or Eager parameter evaluation to implement these two approaches.

7. Action Pot execution can also be executed in Lazy, Eager or Scheduled modes. Current programming languages only use Eager mode. Lazy mode approach is only executed when the result is actually asked for by another operation. Smart mode gives priority to Eager Action Pots and only then execute Scheduled ones, except if new Eager actions are added or Lazy actions are asked.
8. As seen on Data Pots, there are hooks for holding Action Pots when Get and Set operations are called.
9. All Action Pots inputs are passed by value and never by reference. No Action Pot calls Set methods on the Data Pots passed as input. They may create new Data Pots by copy and set values on those.
10. All action Pots can be converted to a sequence than can be transmitted over the network. This ability makes the code movable and enables remote execution. If an action works on a set of huge data that is located in a remote computer, it is easier to transfer the code and execute it there, than to transfer the data over the network.
11. All Action Pots keep logs with all times they were executed and the parameters and results produced. These enable several improvements:
    a) The ability of transforming difficult executions into search problems for all the examples that have already been solved in the past. These have an accumulative result as programs get better with experience.
    b) The ability to know which parameter are more common and from there create automatically generate compiled variants that can be executed on certain conditions on the parameters. These enable programmers to become better over time as they adapt to provide the same results faster based on constraints.
    c) In case of limited combination of possible values, the compiler can even compute in Scheduled mode values in advance to its cache to better serve their caller.
    d) The ability to find patterns between output and input and therefore hint programmers on better ways to solve the problem.

Basic Action Pots should also record their mark in the Provenance of the resulting Data Pots.

7.1.11 Mapping Action Pot

Most of the work in strongly typed languages is actually converting values from one structure to another. Although this work does not argue in favor of strong typing, there is the need of converting Data Pots into the expected format of Action Pots that need to run the date. Therefore, Mapping Action Pot act as adaptors, and there should be visual ways to program the mapping of values between different Data Pots.

7.1.12 Strategy Action Pot

As already mentioned before, Action Pots can be improved automatically to specialized combinations of input parameters. There is also the alternative of the user to try to use a different algorithm to solve the same problem.

There may be cases where on algorithm works better than another. The system can check these situations using as reference the logged calls performed in the past, checking for inconsistencies (eventually bugs) but also for performance.

The same can be applied to an upgraded version of a same function. Be able to assure that it works better for all calls in the past is a great assurance.

7.1.13 Visual Action Pot

Action Pots can be seen as black boxes that solve problems in a magic way. But on the other side, the ability to view an aggregate of all function calls in a visual way could bring insights on bottlenecks and possible improvements. This problem addresses the issue of visibility reported by Frederick Brooks. This is a path that might be explored in future work.

7.1.14 Grammar Action Pot

One of the most constraining issues of programming languages is the strict syntax that is normally required when compared with the flexibility of human language. Calling a function (like calling an Action Pot) is just a way of saying, use this predicate over this input parameters and put the result in that location.

Typical function calls in programming languages look like:

Result := [object.]function_name (parameter 1, …, parameter N)

But the same programming languages realize that this is inadequate for expressing mathematical expressions and therefore create exceptions for the +, - , x, / operations. But not for log, exponential, or squared root operations.

It makes no sense to have this kind on limitations in modern programming languages. It should be possible to extend the syntax of the language to add syntactic sugar to function calls to allow them to be used the the most easy way for the programmers.

For example, a square root of 9 could be expressed as “\/9”, as well as the triple root of 125 should be able to be expressed as: “\3/125”. Likewise, the log base 2 of 8 should be able to be expressed as “log 2 8”. The flexibility of stating the order of appearance of the parameters and the text or symbols that identify the predicate could create more natural ways of stating the instructions.

The ability of the language to be adapted dynamically is a feature that is not common for many languages, but other like Lisp have this feature.

7.1.15 Turing Machine Pot

A Turing Machine is a simple model created by Alan Turing in 1936 that has been used since then to describe all the things that can be computable. Although many alternatives have been proposed, they all are equivalent to a Turing Machine.

A Turing Machine can be informally described as an infinite tape with discrete elements and a machine head at a specific position at each time. The head follows a state machine that based on the symbol currently on the head position and the current state on the state machines decides what to write on that stripe position, where to move next (left or right) and what state to go to in the state machine. The input data is initially placed on the tape and all empty positions are marked empty.

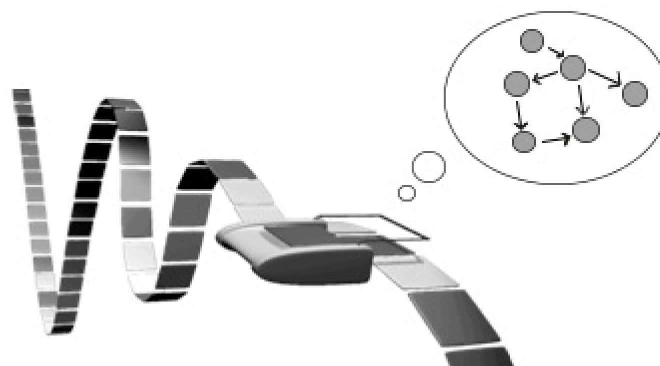


Figure 27 - Visual representation of a Turing Machine

This very visual and simple model has been shown to be enough to describe all possible algorithms. Many advanced Turing machines have been proposed, namely with non-deterministic state machine; with multiple tapes instead of just one; and even probabilistic Turing Machine and Quantum Turing Machine.

In a more forma definition [111], a Turing Machine can be described as

1. A finite set of states that will form the state machine that determines the rules
2. An input alphabet with all the possible values to make changes in the state machine
3. A tape alphabet with all the possible values to be placed in the tape
4. A transition function that defines the transitions between the states of the state machine
5. A start state for the state machine
6. A blank symbol for the empty positions in the stripe.
7. A set of final states in the state machine

What we propose as the tipping point in this work is to have a Turing Machine Pot that:

- uses Data Pots as the elements to be placed in each position of the tape with all the flexibility to handle ambiguity previously described for this artifact and to handle complex data structures
- to have a non-deterministic state chart as the possible representation for the state machine for the head
- to allow multiple stripes at the same time, namely one for input, one for internal processing and one for output (and eventually more)
- to have flexible moving positions in the tape (move N positions left or right, directly access a position; go to the next empty position, etc.)
- to have multiple heads – representing the several cores in a computer
- and to have Action Pots (with all its expressive power previously presented) as the transition functions for handling each input for each head depending on the current state in each core.

This very powerful and visual model that should be able to model any algorithm for a single and for a multi-core computer. We believe it could be the universal architecture for problem solving using computation with the advantage of having a obvious visual representation, addressing the visibility problem identified by Frederick Brooks [15].

The ultimate step would be the construction of a Universal Turing Machine Pot [112], that like the Java Virtual Machine loads and runs programs in Java, would load Turing Machines, check their validity, simulate it, and validate it and execute it with a specific input.

7.2 Method: Formalization Ladder

The purpose of this section is to show how all concepts previously shown can be joined together in order to provide users with a way to incrementally formalize the information's system for its organization. This formalization can happen in a sequence of simple steps.

Following the general idea that software for organizations should grow in an incremental way instead of being build upfront we defined a formalization ladder with 8 steps.

7.2.1 Step 1 - No Formalization (big ball of mud)

In the beginning there is no formalization. Persons send messages (personally, or through email, SMS, social networks, etc) to each other and process requests without any formalized process.

Requests, promises, execution, statement and acceptance are done informally. The provider has only one single transaction to execute: manage messages. Each request is handled differently depending of the request being made. No records are kept.

7.2.2 Step 2 - Adding Dimensions with neurons

Assuming that the provider is using TOPO, he can start using Zachment Framework dimensions to take notice of simple facts in each message he receives from clients, and mark small portions of it as pieces of information just stating the dimensions: who, what, where, when. With this information's, records can be kept. There can be many instances of the same dimension in the same message.

It's as if the client was using colorful markers of the same dimension in the same message.

The provider does not have to provide any additional information, although some notes may become handy in each association.

Neuron Model usefulness

Each relation between the dimensions of Zachman framework an a message (or part of it) is stored using the neuron model.

With this primitive data system within TOPO, the provider can perform simple search based on a time line (when), on a map (where), on a person organization (who) and on assets names (what). We are assuming that TOPO is smart enough to: understand references to dates, knowing the present date and understanding the text; use controlled vocabularies for location differentiating geographical locations from logical ones (within the organization); group names by similarity and parts of names. Names of assets are more difficult to handle at this stage, but they can be handled as tags.

Even with this very primitive information system, automatic data mining is already possible since when is a interval type of data and all the other are categorical data. Therefore it is possible to count how many times a clients performs orders, from where and detailed analysis on when.

7.2.3 Step 3 – Adding Attributes

As the organization flourishes, the provider may feel the need to set roles for specific dimensions, namely create specific tags for each product/services being provided. He may need to record the request time, delivery time, request location, delivery location, etc. He may need to identify which client is the payer, the chooser, the receiver, the beneficiary, the influencing persons to but, etc.

7.2.4 Step 4 – Adding Concepts

At some point in time, organizing all information around the order message is no longer the most efficient and logic way to proceed since is too much repetition of data. There is the nedd to add concepts: client, delivery location, resources, equipments, etc.

However, TOPO still allows the user to freely associate as many Zachman dimensions as needed to each record of each concept. Keeping this flexibility allows to still have the benefits described in steps 2-4, even for the new concepts.

7.2.5 Step 5 – Adding Constraints

In order to prevent data records with invalid data, some field may require to set a list of possible values. Some might even be associated with controlled vocabularies defined by the user. Some attributes may be marked as mandatory (items in order and amount), others as recommended. These constrains of possible values can be formalized as new concepts called types.

In TOPO concept types are described using the descriptive metadata structure. They allow to name an attribute, contrain possible values (either categorical, ordinal, interval and ratio),

describe it, associate a unit of measure (for automatic conversion) and an associated error in measurements (for error propagation calculus).

Numeric attributes are the usual victims of constrains since they are more prone to errors and more difficult to detect and have more variants on format and usage.

With concepts, attributes and types we have the usual building blocks normally used in developing software. Notice that with these new information a wider range of data mining can now be performed automatically. Many patterns should be provided by TOPO in order to facilitate creation and transformation of structural metadata.

7.2.6 Step 6 – Adding Structural Relationships between concepts

With the new concepts and attributes comes the need of structural metadata. Structural metadata allows to model in TOPO UML or Entity Relationships or State Model from DEMO.

By supporting all types of structural relationships according to some foundational ontologies, the autors expect to provide a more powerful solution than the common ones.

It should be possible to add, remove and transform concepts, attributes and their structural relationships over time with bounded effects as prescribed by normalized systems.

7.2.7 Step 7 – Adding Transactions

DEMO methodology introduces a pattern of coordination acts and the ability to set dependencies between transactions on certain coordination acts.

TOPO will not commit to a unique possible pattern, as some difficult issues still exist like the need for the client to be the initiator of the transaction, the difficulties in the integration of infologic and datalogic transactions in the ontological pattern, the handling of delegation and the handling of discussion states after cancellations.

Each of the four types of acts (coordination, production, knowledge and meaning) can have specific interfaces to help users specify their operations. Some interfaces are just plain input fields in a form format. Others need to provide many information in structured visualization formats in order to the user to pass the instructions of what he requires to be done over specific Data Pots. An interface, no matter how simple or how complex is just a combination of input parameters to an Action Pot – or using the traditional terms, a call to a function.

Formalizing tasks in the action model

Organizations have to adapt to circumstances in a rapid changing environment. Therefore users with the appropriate level of responsibility should be able to modify the tasks of a specific action model. This should allow formalizing what should be done typically, what should be checked for a set criteria. But it should also be override be a responsible person that is able to evaluate that a different procedure is better suited for that situation.

The neuron model can be used to setup expressions or even ser as basis for neural networks to classify clients and business opportunities based on past experiences.

7.2.8 Step 8 – Delegating tasks

The final steps consist on persons delegating some responsibilities on software agents (more or less intelligent) that act on the persons behalf.

7.3 Analysis of Ladder Formalization

The usual way software is developed today is jump directly to steps 5 to 7 – restraining fields and data types, and concepts to hold them. Most of the times software development doesn't support transactions at its full extent (steps 8-12), but also blocks the possibility to store non structured data, as in steps 2-4.

In the authors opinion, this lack of flexibility creates more pressure to change software as all small (even if infrequent) differences from expected path creates a barrier for the users because they normally do not allow any walk around solution.

7.4 Prototype - Interface

The prototype image below show the agenda window with the messages to be handled. On the left side of the screen you can see a grid based on Convey's Matrix that has importance on X-axis and Urgency in y-axis. Both dimensions have only 3 values (low, medium and high) but they are represented using the Golden Ratio, which makes high urgency and high importance actually occupy 25% of the screen size, while low urgency and low importance only 4%.

Each dot in the box represents a message that may be in gray if not handled, in green when done, in orange if it was postponed and in red if it was canceled.

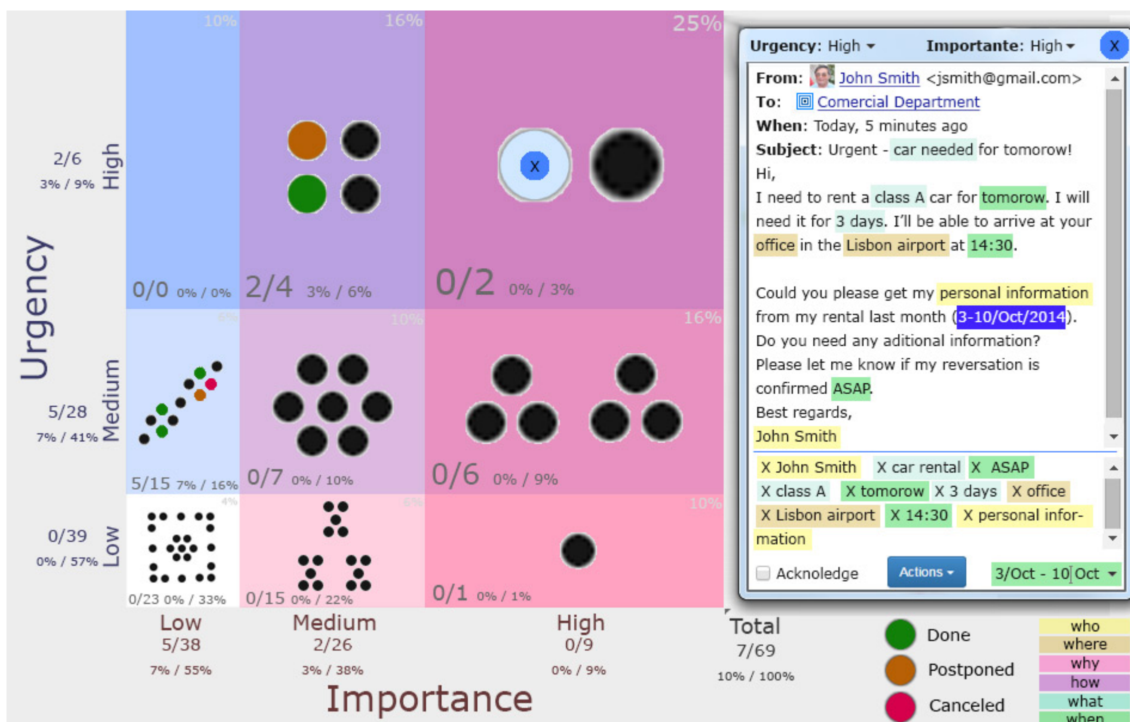


Figure 28 – Prototype for message handling and association of Zachmann Dimensions

The dots presented in each cell of Convey's Matrix follow a pattern according to the prime factorization of the number, as can be seen in the image below.

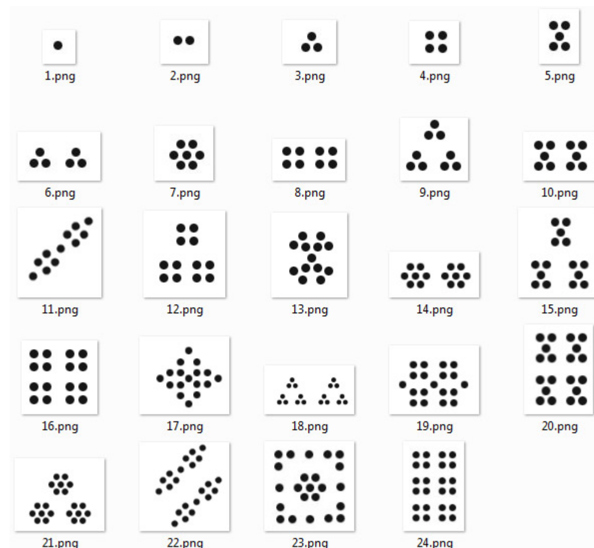


Figure 29 - Distribution of Dots according to prime numbers

On the right side of the Figure 28 we can see one of the messages to be handled (the one in cyan). On the top bar we can change the priority assign to this message either in urgency or in importance. The agent can automatically assign importance and urgency based on the persons involved and the current transactions open with that person.

With the bottom options, the user can acknowledge the message, as well as perform actions on that message like Postpone or Cancel, but also add dimensions with colors for the selected part of the message. There are 5 distinct colores for who, where, why, how, what and when. This is the basic classification that is described in the step 2 of the formalization ladder.

In the Figure 30 we can see a prototype for a message to handle Step 4 – Adding Concepts. The figure shows a list of records of the concept person. There are several ways to search (simple search, advanced search querying for specific values in specific fields), saved resultsets and saved queries. When a result set is obtained it is possible to filter the results or select the one that interest the user. Then it is possible to edit all selected results, create copies, print or delete them.

Who What Where How Why When Config Help Logout

Persons

Watch searching | Fix cleaning | Learn extracting | Model reasoning | Predict forecasting

Watch searching — Look at what data. What does it tell you?

Search, group, order, filter, check each result, edit, copy, delete, create. There is so much that you can do with your data...

Search

Simple Search | Advanced Search | Saved Resultsets | Saved Queries | Show All | Clear Results

Search

Results

List | Dynamic Table | Dynamic Graph | Timeline | Order by | Group by | Actions | Import | Export

Check All
Uncheck All
Invert All
Check Page
Uncheck Page
Invert Page
Remove Checked from Results
Remove Unchecked from Results

Name	Birthday	Category	Last Access
	1968 Dec 16	B	2014 Out 26
	1981 Jan 17	A	NA
	1973 Jan 25	A	2014 Nov 14
	1964 Sep 06	C	2014 Sep 05
	1935 Mar 18	A	2006 Jun 03
	1982 Apr 15	D	2014 Nov 12
John Bolton	1967 Apr 10	A	2014 Out 07
John Church	1994 May 7	C	2014 Out 12
John T. Endicott	2001 Feb 03	B	2013 Nov 28
John Smith	1974 Jun 12	A	2014 Apr 05

Page 1 ... Page 5 6 Page 7 ... Page 18

New, View, Edit and Delete

New | Edit | Delete | Copy | Print

Figure 30 - Prototype for the handling of concepts (Person)

Figure 31 shows and alternative visualization scheme where Persons are presented in a 2D space after the application of a clustering algorithm for coloring the elements and positioning them on the space according to the proximity each person has to each (either based on neuron information or information on registered transactions).

By selecting one of the dots a window popup with additional information about the user.

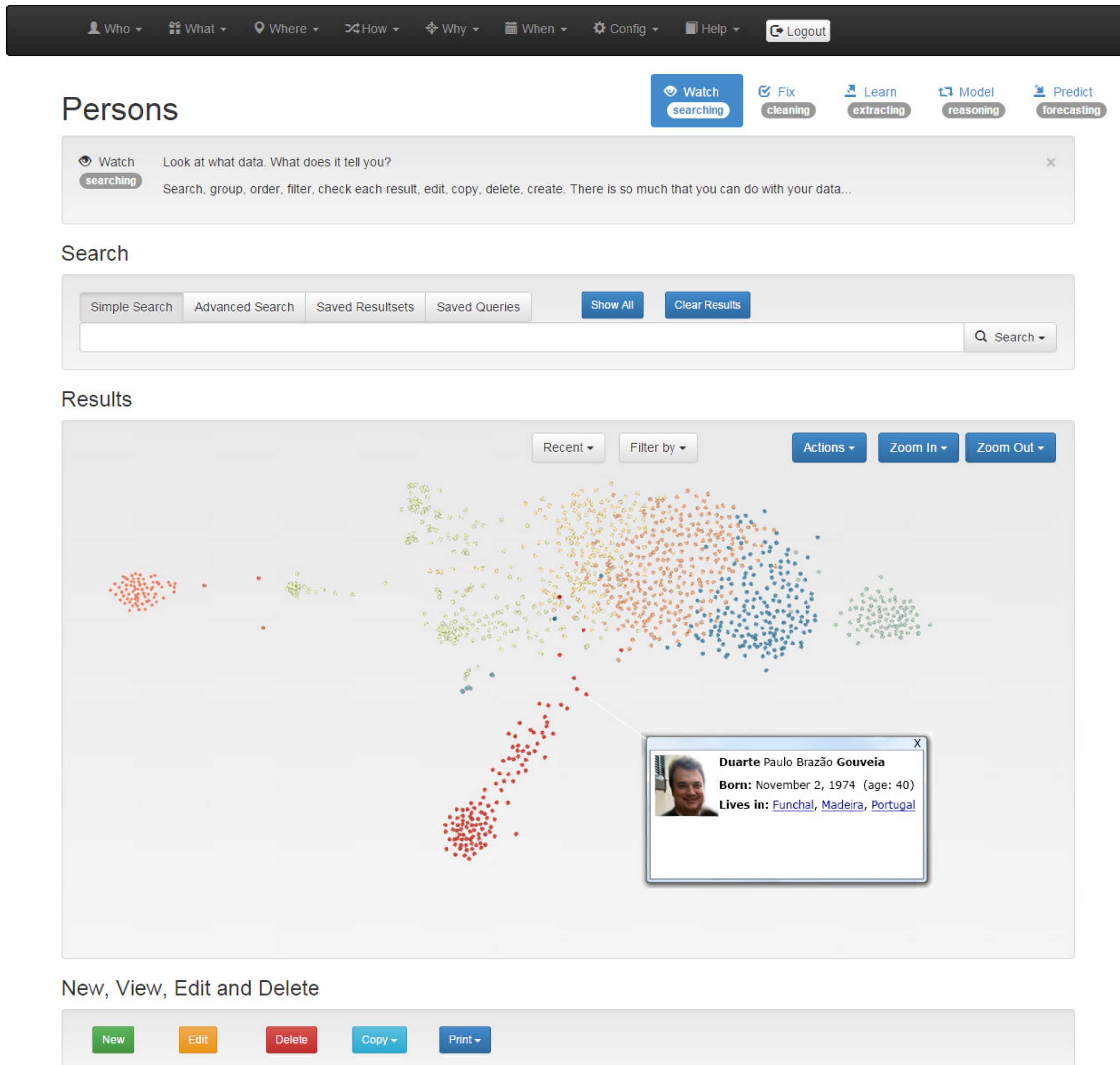


Figure 31 - Prototype for the visualization of complex structures (Clusters of Persons)

There are 5 main views to the system: the agenda of messages to be handled; the visualization of concepts (like persons); the organization unit's hierarchy, and the visualization by transactions (either using the state model or the process state diagram from DEMO).

The main menu also allows navigation be Zachman neuron dimensions: Who, What, Where, How, Why, When. The 5 main views can also be accessed through these menus.

Figure 32 illustrates the navigation window that lets the user select the actor roles and the transaction, in order to get to know who has performed actions as each actor role, what transactions are active, since when and why.

Results

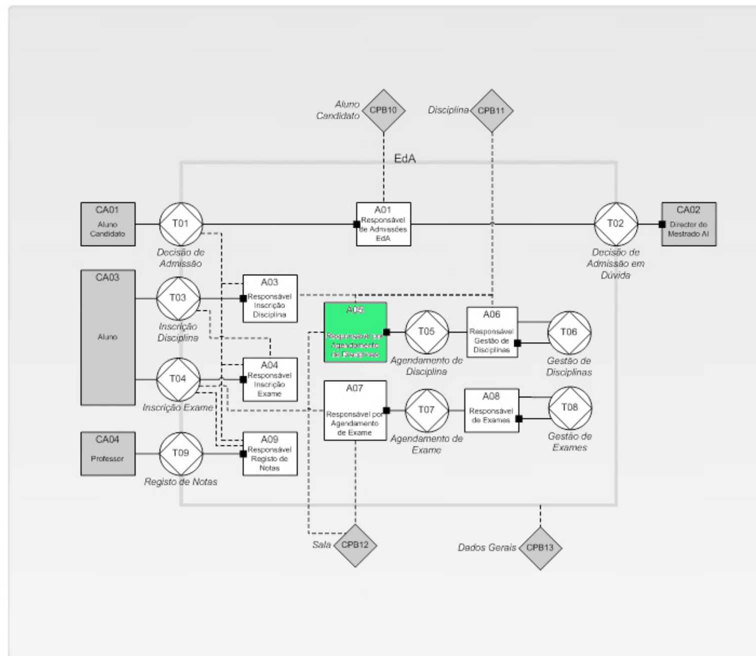


Figure 32 - Prototype for visualization of transactions using DEMO

The prototype figures presented above were constructed as proof of concept for the current architecture and formalization steps.

7.5 Prototype – Inner Structure of Data

The implemented prototype can also be seen at a completely different level than the interface, by looking at a sample of the data structures that build upon each other to implement more complex structures, using the XMF approach of different meta-level architecture presented in section 7.1.2.

In the following sample we start by identifying “identifier”, “boolean”, “letter”, “letter modifier” to then build upon them much more complex structures like “modified letters”, “auto casts”, “regular expressions” and “operators”.

The values presented in pink are lookup values to facilitate the interpretation of the tables. The actual values stored on the tables are the ones in black.

Context: 0

Table: identifier (table key:0 level:0 ordered:0)

fields:

table key types(); Instantiate on level 0;
value types(); Instantiate on level 0;
refers to table types(tables); Instantiate on level 0;
refers to table key types(<-); Instantiate on level 0;

indexes: (empty)
metadata: (empty)
data:

table key	value	refers to table	refers to table key
0	void		
1	identifier	0 identifier	
2	boolean	1 boolean	
3	false	1 boolean	0 0
4	true	1 boolean	1 1
5	digit	2 digit	
6	zero	2 digit	0 0
7	one	2 digit	1 1
8	two	2 digit	2 2
9	three	2 digit	3 3
10	four	2 digit	4 4
11	five	2 digit	5 5
12	six	2 digit	6 6
13	seven	2 digit	7 7
14	eight	2 digit	8 8
15	nine	2 digit	9 9

16	hexadigit	3 hexadigit	
17	vowel	4 vowel	
18	consonant	5 consonant	
19	letter	6 letter	
20	letter modifier	7 letter modifier	
21	character	8 character	
22	invisible character	9 invisible character	
23	modified letters	10 modified letters	
24	auto casts	11 auto casts	
25	category	12 category	
26	category value	13 category value	
27	range	14 range	
28	regular expression item	15 regular expression item	
29	operator	16 operator	

Table: boolean (table key:1 level:0 ordered:0)
fields:
table key types(); Instantiate on level 0;
value types(); Instantiate on level 0;

indexes: (empty)
metadata: (empty)
data:

table key	value
0	0
1	1

Table: letter (table key:6 level:0 ordered:1)
fields:
table key types(); Instantiate on level 0;
value types(); Instantiate on level 0;

indexes: (empty)
metadata: (empty)
data:

table key	value
0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p
16	q
17	r
18	s
19	t
20	u
21	v
22	w
23	x
24	y
25	z

Table: letter modifier (table key:7 level:0 ordered:0)
fields:
table key types(); Instantiate on level 0;
value types(); Instantiate on level 0;

indexes: (empty)
metadata: (empty)
data:

table key	value
0	,
1	~
2	^
3	´
4	`
5	¨

Table: modified letters (table key:10 level:0 ordered:0)
fields:
table key types(); Instantiate on level 0;
value types(); Instantiate on level 0;
letter key types(letter); Instantiate on level 0;
uppercase? types(boolean); Instantiate on level 0;
use modifier? types(boolean); Instantiate on level 0;
letter modifier key types(letter modifier); Instantiate on level 0;

indexes: (empty)
metadata: (empty)
data:

table key	value	letter key	uppercase?	use modifier?	letter modifier key
0	á	0 a	0 0	1 1	3 ´
1	Á	0 a	1 1	1 1	3 ´

2	à	0 a	0 0	1 1	4 `
3	À	0 a	1 1	1 1	4 `
4	â	0 a	0 0	1 1	2 ^
5	Â	0 a	1 1	1 1	2 ^
6	ã	0 a	0 0	1 1	1 ~
7	Ã	0 a	1 1	1 1	1 ~
8	ä	0 a	0 0	1 1	5 ¨
9	Ä	0 a	1 1	1 1	5 ¨
10	é	4 e	0 0	1 1	3 ´
11	É	4 e	1 1	1 1	3 ´
12	è	4 e	0 0	1 1	4 `
13	È	4 e	1 1	1 1	4 `
14	ê	4 e	0 0	1 1	2 ^
15	Ê	4 e	1 1	1 1	2 ^
16	ë	4 e	0 0	1 1	5 ¨
17	Ë	4 e	1 1	1 1	5 ¨
18	í	8 i	0 0	1 1	3 ´
19	Í	8 i	1 1	1 1	3 ´
20	ì	8 i	0 0	1 1	4 `
21	Ì	8 i	1 1	1 1	4 `
22	î	8 i	0 0	1 1	2 ^
23	Î	8 i	1 1	1 1	2 ^
24	ï	8 i	0 0	1 1	5 ¨
25	Ï	8 i	1 1	1 1	5 ¨
26	ó	14 o	0 0	1 1	3 ´
27	Ó	14 o	1 1	1 1	3 ´
28	ò	14 o	0 0	1 1	4 `
29	Ò	14 o	1 1	1 1	4 `
30	ô	14 o	0 0	1 1	2 ^
31	Ô	14 o	1 1	1 1	2 ^
32	ö	14 o	0 0	1 1	5 ¨
33	Ö	14 o	1 1	1 1	5 ¨
34	ú	20 u	0 0	1 1	3 ´
35	Ú	20 u	1 1	1 1	3 ´
36	ù	20 u	0 0	1 1	4 `
37	Ù	20 u	1 1	1 1	4 `
38	û	20 u	0 0	1 1	2 ^
39	Û	20 u	1 1	1 1	2 ^
40	ü	20 u	0 0	1 1	5 ¨
41	Ü	20 u	1 1	1 1	5 ¨
42	ç	2 c	0 0	1 1	0 ,
43	Ç	2 c	1 1	1 1	0 ,
44	A	0 a	1 1	0 0	
45	B	1 b	1 1	0 0	
46	C	2 c	1 1	0 0	
47	D	3 d	1 1	0 0	
48	E	4 e	1 1	0 0	
49	F	5 f	1 1	0 0	
50	G	6 g	1 1	0 0	
51	H	7 h	1 1	0 0	
52	I	8 i	1 1	0 0	
53	J	9 j	1 1	0 0	
54	K	10 k	1 1	0 0	
55	L	11 l	1 1	0 0	
56	M	12 m	1 1	0 0	
57	N	13 n	1 1	0 0	
58	O	14 o	1 1	0 0	
59	P	15 p	1 1	0 0	
60	Q	16 q	1 1	0 0	
61	R	17 r	1 1	0 0	
62	S	18 s	1 1	0 0	
63	T	19 t	1 1	0 0	
64	U	20 u	1 1	0 0	
65	V	21 v	1 1	0 0	
66	W	22 w	1 1	0 0	
67	X	23 x	1 1	0 0	
68	Y	24 y	1 1	0 0	
69	Z	25 z	1 1	0 0	

Table: auto casts (table key:11 level:0 ordered:0)

fields:

table key types(); Instantiate on level 0;
from table types(tables); Instantiate on level 0;
from table key types(<-); Instantiate on level 0;
to table types(tables); Instantiate on level 0;
to table key types(<-); Instantiate on level 0;

indexes: (empty)
metadata: (empty)
data:

table key	from table	from table key	to table	to table key
0	1 boolean	0 0	2 digit	0 0
1	1 boolean	1 1	2 digit	1 1
2	2 digit	0 0	3 hexadigit	0 0
3	2 digit	1 1	3 hexadigit	1 1
4	2 digit	2 2	3 hexadigit	2 2
5	2 digit	3 3	3 hexadigit	3 3
6	2 digit	4 4	3 hexadigit	4 4
7	2 digit	5 5	3 hexadigit	5 5
8	2 digit	6 6	3 hexadigit	6 6
9	2 digit	7 7	3 hexadigit	7 7
10	2 digit	8 8	3 hexadigit	8 8

11	2 digit	9 9	3 hexadigit	9 9
12	3 hexadigit	0 0	4 vowel	0 a
13	3 hexadigit	0 0	4 vowel	1 e
14	3 hexadigit	0 0	5 consonant	0 b
15	3 hexadigit	0 0	5 consonant	1 c
16	3 hexadigit	0 0	5 consonant	2 d
17	3 hexadigit	0 0	5 consonant	3 f
18	4 vowel	0 a	6 letter	0 a
19	4 vowel	1 e	6 letter	4 e
20	4 vowel	2 i	6 letter	8 i
21	4 vowel	3 o	6 letter	14 o
22	4 vowel	4 u	6 letter	20 u
23	5 consonant	0 b	6 letter	1 b
24	5 consonant	1 c	6 letter	2 c
25	5 consonant	2 d	6 letter	3 d
26	5 consonant	3 f	6 letter	5 f
27	5 consonant	4 g	6 letter	6 g
28	5 consonant	5 h	6 letter	7 h
29	5 consonant	6 j	6 letter	9 j
30	5 consonant	7 k	6 letter	10 k
31	5 consonant	8 l	6 letter	11 l
32	5 consonant	9 m	6 letter	12 m
33	5 consonant	10 n	6 letter	13 n
34	5 consonant	11 p	6 letter	15 p
35	5 consonant	12 q	6 letter	16 q
36	5 consonant	13 r	6 letter	17 r
37	5 consonant	14 s	6 letter	18 s
38	5 consonant	15 t	6 letter	19 t
39	5 consonant	16 v	6 letter	21 v
40	5 consonant	17 w	6 letter	22 w
41	5 consonant	18 x	6 letter	23 x
42	5 consonant	19 y	6 letter	24 y
43	5 consonant	20 z	6 letter	25 z

Table: regular expression item (table key:15 level:1 ordered:0)

fields:

table key types(); Instantiate on level 0;
or? types(boolean); Instantiate on level 0;
content table types(tables); Instantiate on level 0;
content key types(<-); Instantiate on level 0;
next regular expression item types(); Instantiate on level 0;
min cardinality types(); Instantiate on level 0;
max cardinality types(); Instantiate on level 0;
starting? types(boolean); Instantiate on level 0;
ending? types(boolean); Instantiate on level 0;

indexes: (empty)
metadata: (empty)
data:

table key	or?	content table	content key	next regular expression item	min cardinality	max cardinality	starting?	ending?
0	0 0	8 character	31 |	1	1	1	0 0	0 0
1	0 0	8 character	6 °		1	1	0 0	0 0
2	0 0	8 character	16 <	3	1	1	0 0	0 0
3	0 0	8 character	28 =	4	1	1	0 0	0 0
4	0 0	8 character	17 >		1	1	0 0	0 0
5	0 0	2 digit	1 1	6	1	1	0 0	0 0
6	0 0	2 digit	0 0		1	1	0 0	0 0
8	1 1	15 regular expression item		9	1	1	0 0	0 0
9	1 1	14 range			1	1	0 0	0 0
10	0 0	15 regular expression item		11	1	1	0 0	0 0
11	0 0	12 category	1 card suit		1	1	0 0	0 0

Table: operator (table key:16 level:1 ordered:0)

fields:

table key types(); Instantiate on level 0;
description types(arity); Instantiate on level 0;
category types(category value); Instantiate on level 0;
prefix character table types(tables); Instantiate on level 0;
prefix character key types(<-); Instantiate on level 0;
infix1 character table types(tables); Instantiate on level 0;
infix1 character key types(<-); Instantiate on level 0;
infixn character table types(tables); Instantiate on level 0;
infixn character key types(<-); Instantiate on level 0;
sufix character table types(tables); Instantiate on level 0;
sufix character key types(<-); Instantiate on level 0;
operand1 type types(tables); Instantiate on level 0;
operand2 type types(tables); Instantiate on level 0;
operand3 type types(tables); Instantiate on level 0;
operandn type types(tables); Instantiate on level 0;
result type types(tables); Instantiate on level 0;

indexes: (empty)
metadata: (empty)
data:

table key	description	category	prefix character table	prefix character key	infix1 character table	infix1 character key	infixn character table	infixn character key	sufix character table	sufix character key	operand1 type	operand2 type	operand3 type	operandn type	result type
0	conjunction	0 unary	8 character	30 !							1 boolean				1 boolean
1	conjunction	1 binary			8 character	21 &					1 boolean	1 boolean			1 boolean
2	dijunction	1 binary			8 character	31 |					1 boolean	1 boolean			1 boolean
3	exclusive dijunction	1 binary			15 regular expression item	0 0					1 boolean	1 boolean			1 boolean
4	implication	1 binary			15 regular expression item	3 0					1 boolean	1 boolean			1 boolean
5	equivalence	1 binary			15 regular expression item	2 0					1 boolean	1 boolean			1 boolean

8. Functionalities and Requirements Analysis

This section contribution:

Analytical evaluation how each requirement proposed in section 5, is addressed by the solutions proposed in sections 6 and 7, and its level of success in complying with those requirements.

R01. Have an infrastructure to support the system.

TOPO is the global architecture for the system and Data Pot and Action Pot the building blocks that are able to represent from the most basic element to the most complex. The architecture sounds valid, complete and robust.

R02. Handle any amount of data.

The usage of a global memory system and an infinite system for keeping references to concepts and instances allows the handling on any amount of data. It is described the automata for keeping the data consistent, but it is not clear if that model would work with the delay and the jitter in a real environment over the Internet.

R03. Handle any type of data.

Data Pots can handle the usual basic data types and also complex data structures. However there was not enough detailed description on how to handle complex sequences of special types of data like audio, video and other special formats.

R04. Handle any number of software applications.

The architecture seems to be able to handle any number of applications. However, It is not clear in the architecture how each application keeps its data apart from other applications, nor how new versions of Action Pots are introduced.

R05. Handle software applications of any size.

The global arquitecture handles the application size without any problem.

R06. Handle any number of stakeholders.

The problem of different stakeholders having different perspectives on the same organization is not properly addressed by the global architecture, as there is no easy to assure synchronization and consistentency between different models.

R07. Handle latency and jitter in communications.

The use of Enterprise Integration Patterns, message queues and an asynchronous model seems to be enough to assure good handle on latency and jitter.

R08. Support multi-strategy algorithms.

Action Pots support multi-strategy pattern.

R09. Support parallel algorithms.

Turing Machine Pot with the proposed extensions seems to be a powerfull model to support parallel algorithms, but needs to be tested.

R10. Support multi-paradigm programming.

The possibility of eager, lazy and smart execution and evaluation of input parameters allows the implementations of many programming paradigms. The use of PSI theory transactions enables the design by contract paradigm. There may be paradigms that are nor supported, namely formal deductive mathematical paradigms.

R11. Support many models to explain and predict events/behaviors.

The flexibility of Data Pots and Action Pot, namely the adoption of multi-value, weights and constrains, combined with the several execution and simulations strategies (like random simulation with tabula rasa rule) allows the representation of many models. Neural network models and Bayesian networks were not addressed.

R12. Support many patterns and their combination on pattern languages.

The combination of patterns into pattern languages was not properly addressed by this architecture.

R13. Remember everything (events) in the system, unless instructed to forget.

Remembering everything is properly addressed. The mechanism about forgetting is not clear enough, neither is the data structure to handle the information of who knows what that is so common in social interactions.

R14. Reuse the same code with minimal maintenance cost (build to last).

Action Pot are build to last, but the actual way that change is performed on Action Pots is not clear.

R15. Provide real time response within specified constraints for real time systems.

Action Pots are able to give intermediary responses. The actual mechanism to assure a response in a certain time frame is not yet clear.

R16. Be as fast as the supporting technology allows it (including parallel environments).

Action Pots address this issue by implementing the strategy pattern that would prevent some code alternatives that use external action that are not available. However this is not automatic, so the requirement is not fully guaranteed.

R17. Continuously improve performance by increase knowledge of system and users.

Action Pot architecture address this issue, by keeping logs and cached results, allowing the automatic creation of more efficient code variants for the most common inputs.

R18. Allow periodic code re-regeneration based on the new versions of used actions and overtime advanced compilation to memory for finding optimizations in the functions where usage justifies the optimization effort.

The Action Pot architecture addresses this issue, but the actual implementation might be challenging.

R19. Handle centralized and distributed data consistently

The architecture addresses this issue, favoring centralized data with an automata to determine which entity is authorized to read and write, and who is the owner.

R20. Facilitate the input and output of data with external systems

The architecture addresses this issue with a special agent that would handle the API. However not sufficient detail is given about this topic.

R21. Support complex data structures in system infrastructure

Data Pot handles complex data structures in a very elegant way.

R22. Support several layers of abstractions to manage complexity

TOPO global architecture addresses this issue creating layers for the interfaces, gesture detection, agent, data and action pots within applications.

R23. Perform forward inference over events to provide timed analysis

Action Pots address this issue, however it is not clear exactly which data structures are more adequate to hold results and enable timed analysis.

R24. Allow temporal analysis for specified or implicit/typical time ranges.

The Time Ontology used is particularly efficient in addressing this issue.

R25. Allow to travel in time in the system, that is, perform (data and actions) as if the system was in a certain date in the past to perform analysis in scenarios.

The system architecture addresses this issue. Concrete implementation is needed to analyze its usefulness.

R26. Allow to simulate future scenarios based on current knowledge and combinations of minor changes in derived variables.

This was not addressed by the current architecture.

R27. Allow the system to evolve in bounded ways that minimize the cost of change without “aging” the code.

Data Pot and Action Pot version control enable this controlled evolution with the ability to do automatic testing with previously used real data.

R28. Assure minimal and identified dependencies with external constraints.

External Action Pots allow this requirement to be addressed.

R29. Certify specific requirements in systems or in program code, either by the process implemented or by statistical analysis of runs over realistic scenarios.

This was not address by current architecture.

R30. Provide means to inspect quality, either automatic or through user collaboration and report them using metrics.

The logs kept by each Action Pot allow the automatic quality inspections and metrics.

R31. Manage rules, or preferences, that can change over time, and optimize the system to comply with the current rules and preferences.

These configuration rules were not addressed by the current architecture.

R32. Allow the robust recovery from a wide range of disaster scenarios.

Logs of operations should allow recovery, but the issue of replication of logs was not addressed.

R33. Allow control actions over registered events and take appropriate actions.

Data Pots have hooks where control actions can be attached for the set and get operations.

R34. Be automatically resilient and fault tolerant to changes in the environment by automatically trying to find alternative ways to reach the same goal and reaching for system administrative help otherwise (for instance: less memory usage, less communication, less disk space, less parallel processors, faster results with less precise results – in general less resources or less time.

This was not properly addressed at this moment by the architecture.

R35. Allow the execution of code either locally or remotely regarding the best interest of the system.

Action Pots can be streamed to allow remote execution. The getting back with results was not properly addressed as it may require a specialized Action Pot.

R36. Expand or contract the level of services according to preferences with minimal deployment cost.

This was not addressed by current architecture.

R37. Allow diverse ways of interacting with the system.

Interface models allow different channels adapted to each context.

R38. Take advantage of localization to customize interfaces according to the user preferences.

The separation between data models and interface models allows handling customization to each user preferences.

R39. Adapt to constraints and have graceful degradation with increasing limitations.

Interface models allow this kind of adaptation. However there needs to be mechanisms to detect that the conditions are degrading which was not addressed.

R40. Run in a new environment with bounded number of adaptations.

The usage of External Data Pots address this issue.

R41. Allow to visually check the system and instantly understand its needs and its strong points at each moment in time.

This was not addressed by current architecture.

R42. Handle many different interfaces with stakeholders, each of them with a specialized data broker that makes data available to the interface in its proper context, in a timely manner and with the best possible visual language and attractiveness for that user and environment (including accessibility constraints)

This must be demonstrated by a fully working prototype, but the concerns were addressed by the architecture.

R43. Provide self teaching interfaces, at least for the first time users of the system.

This was not addressed by current architecture.

R44. Provide multilingual interfaces as a standard feature of the system.

Interface models partially address this issue, but not fully.

R45. Provide appropriate documentation in many formats using single source methodology and being mostly generated in semi-automatic ways.

This was not addressed by current design options.

9. Analysis using Design Science Research Guidelines

This section contribution:

Analysis of current work using Design Science Research Guidelines

Guideline 1: Design as an Artifact

This work is directed to the construction of powerful building blocks (Data Pot and Action) that are coherently combined to create a fully working artifact. These constructs introduce many innovative and powerful ideas that may effectively accomplish the desired effects. The proposed solutions uses solid constructs that are based on sound mathematical theories, although not commonly used in the development of information systems.

This work also proposes a method – the formalization ladder - to allow organizations to gracefully improve their information systems gradually over time, only formalizing what really benefits with formalization, and at the same time keeping the mechanisms (neuron network structure) to overcome unexpected situations that were not handled by the current formalization.

At this time, the presented prototypes are only proof of concept, not yet ready for common use for any organization. Current prototypes support the idea of feasibility of the proposed solutions.

Guideline 2: Problem Relevance

The difficulties in handling **change** in organization's information systems is a relevant, unsolved and difficult problem as described in section 3 – problem statement. The proposed solution not only addresses the technology based artifacts needed to address the problem, but also the organization approach to manage the social system based on communication and consensus building.

Guideline 3: Design Evaluation

The quality and efficacy of a design artifact can only be rigorously evaluated with tests over a full implementation, which was not yet accomplished. However, the evaluation criteria used, based on critical analysis on a list of 45 pre-defined non-functional requirements is a good starting point. We are fully aware that the complexity of the expressed non-functional requirements are too demanding for the current state of the art in information system development, but guidelines for future systems.

Most of the desired features hereby expressed cannot be evaluated with simple controlled experiments, as they lack the global context of applicability. Most of them are engineering concerns that only become apparent when the need for change in organizations becomes effective.

Due to the novelty of many of the proposed solutions in this work, we used descriptive method and static analysis on the architecture properties, as forms of evaluations were not feasible within the time constraints for this work.

Guideline 4: Research Contributions

In this work we introduce many novel and significant research contributions that propose new approaches to difficult problems. Since this is also the conclusions to this work we list the detailed contributions in section 0 - Conclusions.

Guideline 5: Research Rigor

Research rigor is obtained by the use a systematic methods for constructions and evaluation, based on sound and solid theoretical foundations and research methodologies.

We believe this document shows that this approach was performed with those appreciated properties by providing the supporting math, requirements and evaluation with constraints.

Guideline 6: Design as a Search Process

Designing such a complex architecture such as the one hereby described in an iterative and long process of search for the better and simpler solutions. Even at the design phase, finding the simpler solutions take much effort.

Guideline 7: Communication of Research

In this document we tried to present enough detail for both a technological audience, but also for a management oriented audience. Therefore we avoided the use of acronym, promoted figures to illustrate concepts and tried to use simple language, to be accessible to a broader audience than just a technological focused one.

10. Conclusions

This section contribution:

The lessons learned from this work.

The major contributions from this work are:

1 The formalization ladder for organizations was also proposed with a clear method to allow organizations to grow their information system as needed and when needed, preventing the problems of big design up front and rigid formalizations that are not easy to learn and become a problem to rapid change.

2 This work introduces an holistic approach combining substantial experience from organizations real life difficulties with philosophical, ontological, ideological and technological research into a proposed solution. From this knowledge we created a list of 45 demanding requirements to serve as an evaluation reference for this architecture and future work.

3 The introduction of a neuron model using Zachman framework dimensions (who, where, when, why, how, what) that provides the flexibility to organizations have some of the benefits of a complex information system without the need of too much formalization that raise walls against unexpected situations.

4 The design of a new state chart diagram that models how people can reach consensus using communication, using "tell and sell" approach with acknowledgments that improves many of the difficulties that the DEMO transaction pattern has like: forcing a full agreement before request; not properly handling cancelations; forcing fixed roles of proponent and executor; etc. With this new approach, we can still fully model DEMO transactions, but we have much more expressive power to build networks of transactions.

5 We introduced new types of acts besides production and coordination acts, like meaning acts and knowledge acts. These new kinds of acts are useful to register reasons and knowledge within the transaction patterns.

6 A top-down architecture was proposed, called TOPO that addressed the challenges of new interactions interfaces, proposing an architecture for combining them into a coherent structure of multi-channel input/output scheme. A new approach for the use of agents as sole representatives of persons within the information system, centralizing the information for several applications, and serving as a input/output mail center was also proposed.

7 A global memory system was proposed with a novel indexing system for concept plus instantiations that can grow into infinite as needed.

8 A bottom-up detailed architecture was proposed, based on deep analysis on the nature of data, and combining symbolic representation for strings and a novel multiple representation for numbers based on prime numbers.

9 Data Pots powerful structures were proposed, bringing a way to handle ambiguity, uncertainty, definition of constraints and weighted multiple value storing into a basic building block that had controlled interfaces and evolvability as prescribed by normalized systems.

10 A universal construction was proposed to describe all data structures like lists, stacks, trees, graphs and all their variants with a common structure, as well as a flexible indexing structure that allowed arrays, sparse arrays, associative arrays, mappings, hash structures, all included in a Structured Data Pot that can be used in a building block.

11 From a software engineering perspective this work combined information from the inner architectures of CPU's, with the application of the state transition diagram that handles consistency between caches in a multi-core environment; passing through to the current

available assembler instructions; moving up to programming paradigms and complex data structures and architectures. Being able to bring together such a broad knowledge into a coherent structure that takes advantage of lower level architectural features to support high level purposes.

12 We introduced the closure of Turing Machine Pot that combined many of the previously described items into a power and visual structure that may have the ability of express all computable algorithms as Turing Machines do, but in an efficient way.

13 Action Pots also added patterns and deep functional programming perspectives to a normalized system building block with many software engineering features that may increase testability, evolvability and performance.

14 Substantial mathematical foundations were used as support for the proposed solutions, namely through the use of universal algebras, graph theory, automata, set theory and Turing Machine theory. The proposed solution combined many of this diverse research fields in a novel way.

As a complement for the conclusions already presented in sections 8 and 9, we would like to add that this work is its initial steps. The author choose to propose a global architecture with a challenging set of requirements because we believe it is possible to transform information systems ontology's into working prototypes in an automatic way.

Certainly many constraints and insights will come up from the future steps of full implementation of this architecture.

11. Time and Other Constraints

This section contribution:

This work limitations.

Things take much more time than we would like they should. There is always an infinite level of detail that makes every step into the unknown a significant endeavor.

This work is not close to finish. Not only the full implementation of current architecture is a significant effort that could hardly be done by a single person, but also in architectural terms, the non-functional requirements evaluated in section 8, show that only 25% of them are assured by current architecture options, and other 50% are partially satisfied. They are still open research questions as we, as a community, still don't know how to build information systems that could satisfy all those requirements.

12. Ongoing and future work

This section contribution:

Possible alternatives as future work.

The current architecture introduced powerful mechanisms to handle ambiguity. However there aren't many problems in the literatures expressed in a way adequate to be solved taking advantage of ambiguity. Probably the best source of problems to show these tools in action are from logic and math puzzles.

Structured Data Pots introduce a universal data structure to organize all known data structures with a single way of interaction and map them with indexing data pots. This should enable the representation of even more complex structures with compact representation. A future path of work is to find in the literature problems with difficult data structures and use the developed mechanisms to create better algorithms than the existing ones.

The possibility of compiling to memory allows the implementation of complex optimizations for compilation that typically are not worth the effort in compile time for disk. Knowing which action pots are more requested and problematic in terms of execution time allow us to direct our efforts to specific problems and try to maximize the compiling optimizations for that limited problem.

Besides handling change, the problems of complexity, visibility and conformity also pose major challenges to the development of information systems. Addressing all this problems at the same time introduces a level of complexity that makes everything harder, but at the same time addressing only one part of the problems doesn't actually solve the difficult task ahead.

In general, this works opens many threads of future work. Our main effort was to bring analysis of state of art of many serious problems in Information Systems and present an approach of how these problems could be handled in a different way.

13. References

[1] Toffler, A. (1981). *The third wave* (pp. 32-33). New York: Bantam books.

[2] Scott, W. Richard1987 Organizations: Rational, Natural, and Open Systems.Englewood Cliffs, NJ: *Pren-tice-Hall. ScottOrganizations: Rational, Natural, and Open Systems1987.*

[3] Daniel McFarland (2013). Organization Analysis - Stanford University https://www.coursera.org/course/organalysis (last visited December 17, 2014)

[4] Dietz, J. L., Hoogervorst, J. A., Albani, A., Aveiro, D., Babkin, E., Barjis, J., ... & Winter, R. (2013). *The discipline of enterprise engineering. International Journal of Organisational Design and Engineering*, 3(1), 86-114.

[5] Rolf Strom-Olsen (2014) Critical Perspectives on Management – IE Business School https://www.coursera.org/course/criticalmanagement (last visited December 17, 2014)

[6] Klein, H. K. (2003). Crisis in the IS Field? A Critical Reflection on the State of the Discipline. *Journal of the Association for Information Systems*, 4(1), 10.

[7] Avison, D., & Elliot, S. (2006). Scoping the discipline of information systems.*Information Systems: The State of the Field*, 3-18.

[8] Benbasat, I., & Zmud, R. W. (2003). The identity crisis within the IS discipline: Defining and communicating the discipline's core properties. *MIS quarterly*, 183-194.

[9] Galliers, R. D. (2003). Change as crisis or growth? Toward a trans-disciplinary view of information systems as a field of study: A response to Benbasat and Zmud's call for returning to the IT artifact. *Journal of the Association for Information Systems*, 4(1), 13.

[10] INTENT, C. (2006). The Artifact Redux: Further Reflections on the 'IT'in IT Research. *Information Systems: The State of the Field*, 287.

[11] Lyytinen, K., & King, J. L. (2004). Nothing at the center?: Academic legitimacy in the information systems field. *Journal of the Association for Information Systems*, 5(6), 8.

[12] Kuhn, T. S. (2012). *The structure of scientific revolutions*. University of Chicago press.

[13] Dietz, J. L., & Hoogervorst, J. A. (2011). Enterprise engineering manifesto. *Advances in Enterprise Engineering I. LNBIP, 10.*

[14] Dietz, J.L.G. (2006) Enterprise Ontology – Theory and Methodology

[15] Brooks Jr, F.P. (1987). *No silver bullet – essence and accidents of software engineering*. IRRR computer, 20 (4), 10-19.

[16] Brooks Jr, F.P. (1995). *The Mythical Man-Month, Anniversary Edition: Essays on Software Engineering. Pearson Education.*

[17] http://en.wikipedia.org/wiki/List_of_cognitive_biases (last visited December 17, 2014)

[18] http://www.ted.com/talks/tali_sharot_the_optimism_bias (last visited December 17, 2014)

[19] http://www.ted.com/talks/dan_ariely_on_our_buggy_moral_code (last visited December 17, 2014)

[20] Foote, B., & Yoder, J. (1997). *Big ball of mud. Pattern languages of program design*, 4, 654-692.

[21] Damasio, A. (2012). *Self comes to mind: constructing the conscious brain*. Random House Digital, Inc..

[22] Brabandere, Luc (2014) *On Strategy – What managers can learn from Great Philosophers* – École Central Paris - https://www.coursera.org/course/businessandphilo (last visited December 17, 2014)

[23] Scott Page (2013) Model Thinking – University of Michigan - https://www.coursera.org/course/modelthinking (last visited December 17, 2014)

[24] Mintzberg, H. (1994). *Rise and fall of strategic planning*. Simon and Schuster.

[25] Kaplan, R. S., & Norton, D. P. (2004). *Strategy maps: Converting intangible assets into tangible outcomes*. Harvard Business Press.

[26] Chiang, M. (2012). *Networked Life: 20 Questions and Answers*. Cambridge University Press.

[27] Walter, C. (2005). Kryder's law. *Scientific American*, 293(2), 32-33.

[28] http://www.informationisbeautiful.net/visualizations/million-lines-of-code/ (last visited December 17, 2014)

[29] Hwu, W. M. (2013). Heterogeneous Parallel Programming – University of Illinois https://www.coursera.org/course/hetero (last visited December 17, 2014)

[30] Buxton, J. N., & Randell, B. (Eds.). (1970). *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee; available from Scientific Affairs Division, NATO.

[31] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., ... & Winwood, S. (2009, October). seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (pp. 207-220). ACM.

[32] https://courses.edx.org/c4x/BerkeleyX/CS-169.1x/asset/handouts_slides_169_Lecture1_Course_Introduction.pdf (last visited December 17, 2014)

[33] Lehman, M. M. (1996). Laws of software evolution revisited. In *Software process technology* (pp. 108-124). Springer Berlin Heidelberg.

[34] http://www.php.net/ChangeLog-4.php (last visited December 17, 2014)

[35] http://www.php.net/ChangeLog-5.php (last visited December 17, 2014)

[36] https://bugs.php.net/stats.php (last visited December 17, 2014)

[37] DeMarco, T., & Lister, T. (2013). *Peopleware: Productive Projects and Teams*. Addison-Wesley.

[38] http://www.versionone.com/assets/img/files/CHAOSManifesto2012.pdf (last visited December 17, 2014)

[39] http://www.versionone.com/assets/img/files/CHAOSManifesto2013.pdf (last visited December 17, 2014)

[40] Downey, A. B. (2012). *Think Complexity: Complexity Science and Computational Modeling*. O'Reilly.

[41] Bunge, M.A. (1979). *Treatise on basic philosophy, vol. 4, a world of systems*, Reidel Publishing Company

[42] Cecilia Aragon (2013) Introduction to Data Science –- University of Washington https://www.coursera.org/course/datasci (last visited December 17, 2014)

[43] Bertin, J. (1983). *Semiology of Graphics*, 1967.

[44] Card, S. K., Mackinlay, J. D., & Shneiderman, B. (1999). *Readings in information visualization: using vision to think*. San Francisco, Calif.: Morgan Kaufmann Publishers.

[45] http://langpop.com/ (last visited December 17, 2014)

[46] http://en.wikipedia.org/wiki/Programming_paradigm (last visited December 17, 2014)

[47] http://cs.lmu.edu/~ray/notes/paradigms/ (last visited December 17, 2014)

[48] Parallel Universe Magazine, April/2009 https://software.intel.com/sites/default/files/parallel_mag_issue1.pdf (last visited December 17, 2014)

[49] Abrial, J. R., & Abrial, J. R. (2005). *The B-Book: Assigning programs to meanings*. Cambridge University Press.

[50] Birkhoff, G. (1933, October). On the combination of subalgebras. *InMathematical Proceedings of the Cambridge Philosophical Society (Vol. 29, No. 04, pp. 441-464)*. Cambridge University Press.

[51] Burris, S., & Sankappanavar, H. P. (1981). *A course in universal algebra (Vol. 78)*. New York: Springer-Verlag.

[52] Kunen, K. (2011). *Set theory*. College Publ..

[53] Grue, K. (1992). Map theory. *Theoretical Computer Science*, 102(1), 1-133.

[54] Gross, J. L., & Yellen, J. (2005). *Graph theory and its applications*. CRC press.

[55] Harel, D. (1987). Statecharts: A visual formalism for complex systems.*Science of computer programming, 8(3), 231-274.*

[56] Jeffrey Pomerantz (2014) Metadata: Organizing and Discovering Information — The University of North Carolina of Chapel Hill https://www.coursera.org/course/metadata (last visited December 17, 2014)

[57] http://dublincore.org/ (last visited December 17, 2014)

[58] http://www.france24.com/en/20131101-germany-creates-indeterminate-gender-birth-register/ (last visited December 17, 2014)

[59] Sedgewick, R. (2002). *Algorithms in Java, Parts 1-4*. Addison-Wesley Professional.

[60] Weiss, M. A. (1992). *Data Structures and Algorithms*. Benjamin/Cummings.

[61] Drozdek, A. (2004). *Data structures and algorithms in Java*. Cengage Learning.

[62] http://php.net/manual/en/book.strings.php (last visited January 21, 2014) (last visited December 17, 2014)

[63] Bayer, R., & McCreight, E. (2002). *Organization and maintenance of large ordered indexes* (pp. 245-262). Springer Berlin Heidelberg.

[64] Sipser, M. (2012). *Introduction to the Theory of Computation*. Cengage Learning.

[65] Dietz, J. L. (2006). *What is Enterprise Ontology?* (pp. 7-13). Springer Berlin Heidelberg.

[66] Guizzardi, G. (2005). *Ontological foundations for structural conceptual models.* CTIT, Centre for Telematics and Information Technology.

[67] Gruber, A., Westenthaler, R., & Gahleitner, E. (2006). Supporting domain experts in creating formal knowledge models (ontologies). In *Proceedings of I-KNOW* (Vol. 6, pp. 252-260).

[68] Scherp, A., Franz, T., Saathoff, C., & Staab, S. (2009, September). F--a model of events based on the foundational ontology dolce+ DnS ultralight. In *Proceedings of the fifth international conference on Knowledge capture* (pp. 137-144). ACM.

[69] Grau, B. C., Horrocks, I., Kazakov, Y., & Sattler, U. (2007, January). A Logical Framework for Modularity of Ontologies. In *IJCAI* (Vol. 2007, pp. 298-303).

[70] Gangemi, A., Guarino, N., Masolo, C., Oltramari, A., & Schneider, L. (2002). Sweetening ontologies with DOLCE. In Knowledge engineering and knowledge management: Ontologies and the semantic Web (pp. 166-181). Springer Berlin Heidelberg.

[71] Model, B. M. (2006). Version 1.0. *Standard document URL: http://www. omg. org/spec/BMM/1.0/PDF (22.09. 2009).*

[72] Bittner, T., Donnelly, M., & Smith, B. (2004, November). Individuals, universals, collections: On the foundational relations of ontology. In Proceedings of the Third Conference on Formal Ontology in Information Systems (pp. 37-48).

[73] Mannaert, H., & Verelst, J. (2009). *Normalized systems: re-creating information technology based on laws for software evolvability*.

[74] http://en.wikipedia.org/wiki/Hollywood_principle (last visited December 17, 2014)

[75] Neto, A. B., Gouveia, D., & Silva, M. J. (1998). ACE: um agente de compras na Internet.

[76] Hohpe, G., & Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.

[77] Simmhan, Y. L., Plale, B., & Gannon, D. (2005). A survey of data provenance in e-science. *ACM Sigmod Record*, *34*(3), 31-36.

[78] Moreau, L., Freire, J., Futrelle, J., McGrath, R. E., Myers, J., & Paulson, P. (2008). The open provenance model: An overview. In *Provenance and Annotation of Data and Processes* (pp. 323-326). Springer Berlin Heidelberg.

[79] Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, *21*(5), 61-72.

[80] Fowler, M., & Highsmith, J. (2001). The agile manifesto. *Software Development*, *9*(8), 28-35.

[81] Dijkstra, E. (1968). A Case against the GOTO Statement. Communications of the ACM, 11, 147.

[82] Wulf, W. A. (1972, August). A case against the GOTO. In Proceedings of the ACM annual conference-Volume 2 (pp. 791-797). ACM.

[83] Tufte, E. R., & Graves-Morris, P. R. (1983). *The visual display of quantitative information* (Vol. 2). Cheshire, CT: Graphics press.

[84] Healey, C. G. (2007). Perception in visualization. *Retrieved February*, *10*, 2008.

[85] http://en.wikipedia.org/wiki/Stevens'_power_law (last visited December 17, 2014)

[86] Alexander, Christopher. 1979. *The Timeless Way of Building*. Oxford University Press.

[87] Alexander, Christopher, Sara Ishikawa, and Murray Silverstein. 1978. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. {Oxford University Press}.

[88] Johnson, R., Helm, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. E. Gamma (Ed.). Addison-Wesley Professional.

[89] Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys* (CSUR), 23(1), 5-48.

[90] http://en.wikipedia.org/wiki/IEEE_floating_point (last visited December 17, 2014)

[91] http://en.wikipedia.org/wiki/Unix_time (last visited December 17, 2014)

[92] http://en.wikipedia.org/wiki/French_Republican_Calendar (last visited December 17, 2014)

[93] http://en.wikipedia.org/wiki/Second (last visited December 17, 2014)

[94] http://www.youtube.com/watch?v=gRdfX7ut8gw (last visited December 17, 2014)

[95] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

[96] Shannon, C. E. (2001). A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1), 3-55.

[97] Nicholas James Provart (2014). Bioinformatic Methods I -– University of Toronto - https://www.coursera.org/course/bioinfomethods1 (last visited December 17, 2014)

[98] Benford, F. (1938). The law of anomalous numbers. *Proceedings of the American Philosophical Society*, 551-572.

[99] http://en.wikipedia.org/wiki/Benford%27s_law (last visited December 17, 2014)

[100] http://testingbenfordslaw.com/ (last visited December 17, 2014)

[101] Zhou, Q., & Fikes, R. (2000). *A reusable time ontology*. KSL-00-01, Stanford University.

[102] Linehan, M. H., Barkmeyer, E., & Hendryx, S. (2012, July). The Date-Time Vocabulary. In *FOIS* (pp. 265-378).

[103] Yoder, J. W., Balaguer, F., & Johnson, R. (2001). Architecture and design of adaptive object-models. *ACM Sigplan Notices*, 36(12), 50-60.

[104] David, F. S. (2003). Model driven architecture: applying MDA to enterprise computing.

[105] Clark, T., & Willans, J. (2012). Software language engineering with XMF and XModeler. Formal and Practical Aspects of Domain Specific Languages: Recent Developments. IGI Global, USA.

[106] Guinard, D. (2011). *A Web of things application architecture* (Doctoral dissertation, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 19891, 2011).

[107] History of Operations and System MEMENTO Ferreira, H. S., Correia, F. F., & Welicki, L. (2008, October). Patterns for data and metadata evolution in adaptive object-models. In *Proceedings of the 15th Conference on Pattern Languages of Programs* (p. 5). ACM.

[108] Goguen, J., & Malcolm, G. (2000). A hidden agenda. *Theoretical Computer Science*, *245*(1), 55-101.

[109] Coursera course on Compilers – Alex Aiken – Stanford University https://www.coursera.org/course/compilers (last visited December 17, 2014)

[110] INTEL, INC. "Intel R 64 and IA-32 Architectures Software Developer's Manual."

[111] Jeff Ullman (2014). Automata – Stanford University https://www.coursera.org/course/automata (last visited December 17, 2014)

[112] http://en.wikipedia.org/wiki/Universal_Turing_machine (last visited December 17, 2014)