Cybernetics Oriented Language (CYBOL)

Christian HELLER <christian.heller@tu-ilmenau.de>

Technical University of Ilmenau
Faculty for Computer Science and Automation
Institute for Theoretical and Technical Informatics
PF 100565, Max-Planck-Ring 14, 98693 Ilmenau, Germany
http://www.tu-ilmenau.de, fon: +49-3677-69-1230, fax: +49-3677-69-1220

Abstract

Abstracting the real world is a major aim of Informatics. This paper introduces a new Theory and Language which allow to better abstract the real world in clear and simplified Models than today's software does. It thereby helps crossing a number of abstraction gaps that each software project has to go through in its lifetime.
Sticking to Cybernetics, this paper means that one of the first things to consider for developing good software is how Human Thinking works and how it creates Abstractions. Fundamental principles of human thinking are Discrimination, Categorization and Composition. The abstractions they deliver are Item, Category and Compound. They help the human mind to understand its environment which exists as conglomerate, and to build meaningful models from it.

Keywords. Cybernetics Oriented Programming, CYBOL, CYBOI, Human Thinking, Abstract Model

1 Introduction

One important area the science of *Informatics* deals with is software – the art of *representing* and *processing* information. As such, one of its major aims is to find *Abstract Models* which represent the real world best. The better this is done and the better information can be stored and processed, the better software can assist its human users.

Since about 40 years, the same, often unsatisfying concepts are used in informatics, which caused some people to talk about an ongoing *Software Crisis*. Since about 20 years, the *Free and Open Source Software* (FOSS) Movement increasingly eases that pain by providing a tremendous amount of code containing plenty of new concepts but still – the dream of true componentization and reusability has not been reached. *Structured* and *Object Oriented Programming* (OOP) delivered some new concepts, a major one of was the extension of data *Type* to *Class*, owning inheritable properties and methods. However, there is a number of problems that still keep us away from clear, effective and above all flexible solutions, in particular the:

- false combination and grouping of information
- mix of knowledge and system control information
- bundling of static and dynamic aspects

A more detailed analysis of point one is given in [3]. *Ontologies* are suggested as means for improvement in [4]. They help structuring data models by dividing a domain into singular *Concepts* (as known from *Knowledge Engineering*) which later get associated with each other. Elements of a concept are organized in strict layers which ensures flexibility for later extensions. This paper wants to use those results but also concentrate on point two and three, as listed above, and investigate their negative effects and possible solutions.

As a system grows, the interdependencies between its single parts grow with. Why does this happen? Simply because a clear architecture is missing. Even if developers really try to follow a such – on some point in the software's lifetime, compromises have to be made due to unforeseen requirements and dependencies:

- multiple interfaces are used to realize new properties (Mix-In)
- static manager objects accessible by any other objects in the system are introduced
- new layers are plugged in with varying mechanisms
- redundant code needs to be written to avoid too many unwanted interdependencies

These decisions, in turn, can lead to buggy code with: memory leaks, endless loops, false results, weak performance. Can all that be avoided? And if, then how? The author's opinion is yes and the new concepts and language introduced in this document show ways out of the misery.

2 Software Engineering Process

For a great part, the aforementioned problems are caused by multiple *Gaps* in abstraction, that occure during a software project's lifetime. Software does not only contain and process *Information*, it is information itself. It stands at the end of a sequence of abstractions which is called a *Software Engineering Process* (SEP). Software development history has shown plenty of different forms of such processes, but most can be categorized into one of the following: *Waterfall Process*, *Iterative Process*, *Extreme Programming* (*Cathedral* or *Bazaar* mode), *Agile Software Development*.

This work is not exactly about software engineering processes, nor does it want to introduce yet another one. Its main purpose is to deal with the *Results* of software development phases: *Abstractions*. Probably every project goes through the three common phases *Analysis*, *Design* and *Implementation* (figure 1). Each of them creates its own model of what is to be abstracted in software:

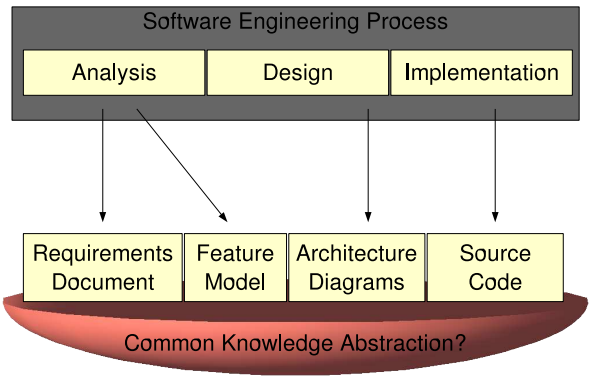

Fig. 1. Knowledge Abstraction

The analysis often results in a *Requirements Document* which investigates the problem domain and uses expert knowledge to specify the functionality of the software to be created. This specification is mostly *informal*, that is an ordered collection of textual descriptions. Sometimes, *semi-formal* descriptions such as tables or graphics are used additionally.

It is the aim of the design phase to deliver a clear system architecture with little redundancies and only few interdependencies, which it may specify by help of *semi-formal Diagrams*. Recent years showed an increased use of the *Unified Modelling Language* (UML), a collection of diagram specifications for representing static or dynamic aspects of a system to be modelled. Normally, a *top-down* approach is chosen for the design of a system. Hereby, the overall architecture is considered first, before moving into details. The less common *bottom-up* design would start the other way and first try to build small components to construct the whole system from.

Finally, implementation of a system is done *formally*, in one (or more) programming languages. The retrieved *Source Code* represents the final abstraction, the software that was to be built.

It is obvious that at least two gaps have to be crossed when using the described phases:

1. Requirements Document – Architecture Diagrams
2. Architecture Diagrams – Source Code

Many efforts try to minimize the first gap by telling their analysis experts to specify use cases, workflows and static structures using the corresponding diagrams provided by the *Unified Modelling Language*. Other efforts introduce more steps of abstraction, like the *Feature Model*. It provides a hierarchical model of the features of the system to be built. The feature analysis is part of the analysis but can logically be placed between analysis and design. It became especially popular in the area of *System Family/ Product Line Engineering*. Yet the disadvantage of using feature models is that another gap in abstraction is created:

1. Requirements Document – Feature Model
2. Feature Model – Architecture Diagrams
3. Architecture Diagrams – Source Code

One aim of the work described in this document was to overcome these gaps by supplying one kind of abstracted knowledge, for statics as well as for dynamics, to be continuously used throughout all project phases.

3 Traditional Programming

Section 2 pointed out a problem that all current software engineering processes are struggling with: *Abstraction Gaps*. To find out about possible reasons, traditional and current programming concepts need to be inspected closer.

3.1 Property Bundling

Software consists of data which can be processed by a computer. This is possible because *qualitative* data are transformed into *quantitative* data and, finally, to *Zero* and *One*. *Mathematics* delivers the *Logic* (*Operations*) after which *States* (*Operands*) can be mapped, to deliver the expected results.

When combining a number of operations in a certain order, an *Algorithm* is retrieved. The operands it works with are stored in *Variables*. As can be seen, there is always *static* and *dynamic* structures involved, the static holding the states and the dynamic holding the rules for mapping between states.

Structured/ Procedural Programming languages were the first to explicitly provide the means to model static *Structures* as well as dynamic *Procedures* (or *Functions*, respectively). Both can be cascaded in *Hierarchies* and even use *Recursion* for that.

Yet other synonyms for static and dynamic structures that were introduced by the nowadays more popular *Object Oriented Programming* (OOP) are *Attribute* and *Method*. Both can be properties of an *Object* which is the runtime instance of a *Class* which in turn represents the data *Type*. A class can *inherit* properties from a *super* class. This *Inheritance* was a truely new and innovative concept brought in by OOP. The *Bundling* of static and dynamic properties (attributes and methods), on the other hand, causes more system interdependencies and complications than were predictable. It is a big disadvantage that affects all modern object-oriented systems.

Certainly, the bundling stems from best intentions to receive cleaner code by keeping not only attributes but also methods in a common module, such avoiding *wild* and *global* procedures. But now, modules not only had to refer to other modules for accessing their data; the same was needed for accessing methods. With OOP, the number of cross-relations between modules and system interdependencies in general nearly always rise dramatically. In reality, static and dynamic properties are two *different* things that have to be kept in different places! Both can have a similar, hierarchical structure but each is a concept on its own.

Interface inheritance is used to implement *Concerns* in a system. *Aspect Oriented Programming* (AOP) calls its concerns *Aspects* and uses special language means to implement them. With standard class constellations, *Design Patterns* provide clear solutions describing how best to combine (associate/ inherit) classes. As subsumption of many design patterns, a *Framework* aims to provide basic functionality for the applications to be embedded into it. All these efforts are based upon OOP. The main idea behind them is to prevent code duplication and to minimize the interdependencies between parts of a system. But what if OOP is one reason for just those interdependencies?

Big systems with a multitude of associations and dependencies often lead to a loss in overview which results in so called *Spaghetti Code*. *Component Oriented Programming* (COP) tries to solve this by encapsulating code in smaller *Components* which shall make it easier to keep overview. People started to dream about a simple combination of such components and called it the *LEGO Metapher* (with relation to the building blocks for children). But software models are not simple building blocks that could be built *Stone-on-Stone*! They are concepts as known from *Human Thinking*.

The components described to here are *passive* because they need a system to call them. More recent studies are about *active* components, sometimes called *Agents*. These are self-acting processes that solve special tasks. The concepts behind are called *Agent Oriented Programming*.

Passive components and all of the programming concepts described before them belong to the *Logical Architecture* of a system. The term *Physical Architecture* is used when it comes to active components, agents, processes or systems in general.

3.2 Information Mix

Of course, the bundling of static and dynamic properties as used in OOP is not the only factor causing interdependencies. Otherwise, traditional procedural programming languages had already delivered ideal systems. But this is not the case. Something else must be missing. The major problem of today's software is its *Mix* of two very different kinds of information: *System Control* and *Application Knowledge*.

A standard computer architecture consists of a *Memory* (which stores data), a *Processor* (that applies operations on the data), *Input/Output Devices* (to correspond with the environment) and a *Bus System* (that connects the beforementioned parts). All these devices need to be controlled in some way. Variable values (instances) need to be written to and read from the memory; operations which the processor offers need to be called; input and output values need to be exchanged through the corresponding input/output devices.

Most of this is done by an *Operating System* (OS) and its hardware drivers. However, programming languages allow their users to access hardware, too. Software programmers can send processor instructions, they can allocate (instantiate) memory etc. It is these possibilities which lead to memory leaks and further software problems. If the operating system, for example, concentrated all memory allocation in one place, forgotten instances would belong to the past.

The remaining code represents the actual *Application*. It contains the *Domain Knowledge*, the *Concepts*, the *Configuration* information. These models of real world phenomenons have nothing to do with hardware control and need to be treated differently.

4 Human Thinking

Criticising *State-of-the-Art* concepts is one part of science; offering improved solutions is its complementary. Researchers quite often follow the approach of first looking into what nature offers and then trying to engineer a similar solution. All kinds of tools and machines were created this way, even (and most obviously, with respect to the human body and mind) robots and computers. Some scientists take the principles of human awareness as physical model to explain the universe [10]. Some business people and consultants see analogies between processes in the human brain and organisational structures of a company [11]. Researchers in human sciences systematise international public law by sharing it into the three parts society, cooperation and conflicts which are chosen in analogy to biology, that is anatomy, physiology and pathology of international relations [1]. Considering all that, one question is at hand:

Why not apply a similar approach to software engineering?

The science of *Cybernetics* and its specialization *Bionics* recommend to *compare* communication and control processes in biological versus artificial systems as well as to *apply* biological principles to the study and design of engineering systems. If computers are built after the model of the human being (information input, memorizing, processing and output), why not structure the software that actually runs those computers after similar models? It seems logical and clear, yet the reality looks different. This section will therefore consider how *Human Thinking* works and how it creates abstractions (figure 2).

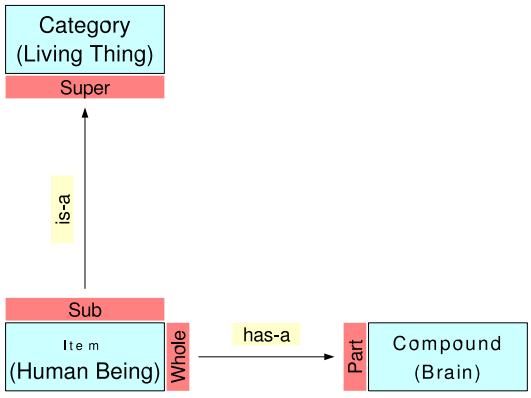

Fig. 2. Human Thinking

4.1 Item

As first and most important abstraction, the human brain divides its real-world environment into discrete *Items*. Physicists call smaller items *Particle*. Plenty of other synonyms exist. Software developers often talk of *Object*. This document preferrably uses the more neutral name *Item*, since models are created not only of objects but also of *Subjects*.

Behavioural psychologists talk of this ability as *Discrimination*. It commonly focuses on a specific real world phenomenon, leaving out parameters which are not interesting in the given context. This is necessary because otherwise, a brain would have to model and capture the whole universe (with every single particle being duplicated), which is obviously impossible. As example, a *Human Being* as item is stated (in parentheses) in figure 2.

Not only human beings, but also some higher animal species (like apes) are able to *discriminate* their environment and to form terms to name it. Additionally, they have a primitive *Self Concept*, that is a term for their own personality. However, their cognitive abilities are limited in that concepts are only available in the presence of the corresponding item. Jaeger [5] calls that *Online Thinking*; cognition scientists speak of *Terms of first Order* or *Sensoric Type of Terms*.

Contrary to this, the more advanced *Offline Thinking* allows humans to think about items they currently cannot sense. Cognition scientists here speak of *Terms of second Order*. They became possible by *associating* sensoric signals with terms of a language. The resulting *Net of Associations* brought a number of advantages [5]:

- *Decoupling* thinking from immediate motoric reaction
- *Time Index* in scenes so that past memories can be recalled, the future be planned
- *Dual Representation* of online and offline contents
- *Self Awareness* thanks to online and offline thinking
- *Associations* increasing the expressiveness of terms

4.2 Category

Offline thinking (in terms of second order) enables humans not only to discriminate items but also to *categorize* them into superior groups. Since it is impossible to exactly model the real world in complete, compromises have to be made: People do not model every single item in their minds but rather group them into *Types* (*Classes*) of common characteristics.

This kind of classification stems from the earliest days of ancient science. *Plato*'s pupil *Aristotle* (being the teacher of *Alexander the Great*) was the first philosopher who logically captured and organized the world. It was him who sorted items into clear groups which he called *Categories*. And it was him who first distinguished between *enlivened* and *unenlivened* nature; who parted living forms into *Plants*, *Animals* and *Humans*. The science of biology calls this classification a *Systematics*.

Categorization (classification) can be seen from two sides, depending on what direction of that relationship one wants to emphasize. Taking Aristotle's examples, *Living Thing* would be a *Generalization* of *Plants*, *Animals* and *Humans*. *Human Being* would be a *Specialization* of *Living Thing*.

Software developers call categorization an *is-a* relationship and talk of *Super* and *Sub* categories (sometimes also *Parent* and *Child* categories). Section 3.1 mentioned that object oriented programming uses categorization to let a sub class inherit attributes and methods from its super class.

4.3 Compound

Composition is the third kind of abstraction that humans use to understand their environment. It is an important instrument for the human mind to associate information, that is to acquire, store and recall *Knowledge*. Every item is recognized as a *Compound* of smaller items and can therefore also be called *Tree* or *Hierarchy*. The subject of *Artificial Intelligence* (AI)/ *Knowledge Engineering* talks of *Concept* or *Schema*.

In software design, the terms *Parent* and *Child* are often used to describe both, the items in a composition as well as the items in a categorization relationship (section 4.2). To avoid misunderstandings, this document sticks to the terms *Super* and *Sub* for categorization and to the terms *Whole* and *Part* [12] for composition. Yet other terms to describe items of a composition would be *Container* and *Element*.

To stick with the example of a *Human Being*, one could say that it is composed of *Organs* such as *Eye*, *Ear*, *Heart*, *Brain*, *Arm* and further, also smaller parts. Other examples are the concept of an *Atom* consisting of a *Core* and *Electrons* or that of a physical *Book* composed of a *Paperback Cover* and *Paper Pages*. However, knowledge representation always depends on what one wants to express in which context. The *Book*, for example, can be represented in many other ways. Logically, it is usually separated into *Part*, *Chapter*, *Section*, *Paragraph*, *Sentence*, *Word* and *Character*.

It is important to note the *unidirectional* kind of relations: A human being is composed of organs but an organ is never composed of a human being!

Not only *static* items represent a compound; *dynamic* items are hierarchical as well. The process *Take Book from Library*, for example, may have the following structure:

- Check Catalogue
  - Investigate suitable Books
  - Note Registration Number
- Organize Book
  - Look for Shelf
  - Take off Book
- Borrow Book

Returning to human thinking, one realizes that in the end, everything in universe can be put into variable hierarchical models, that is consists of smaller items and belongs to a bigger item. From the physical point of view, nobody knows where this hierarchy really stops, towards *Microcosm* as well as towards *Macrocosm*. There is no *absolute*, *basic* item. A *Particle* as concept exists only in the human mind, placed somewhere between micro- and macrocosm, with hypothetic borders.

4.4 Model

A theoretical *Model* is an abstract clip of the real world, and exists in the human mind. Another common word for *Model* is *Concept*. It is the subsumption of *Item*, *Category* and *Compound*, resulting from the three activities of abstraction: *Discrimination*, *Categorization* and *Composition*. As such, each model knows about its super model and the parts it consists of (figure 3). Software developers would call the illustration of these relations a *Schema* or *Meta Model*.

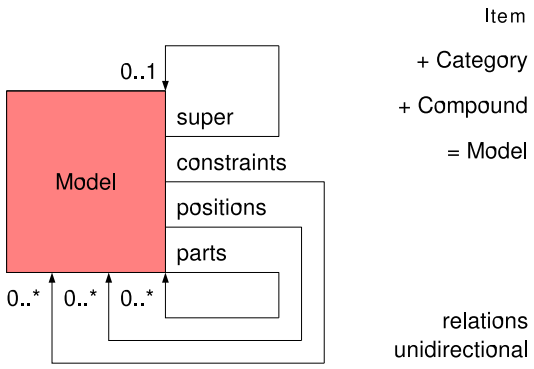

Fig. 3. Model as subsumption of Item, Category, Compound

4.5 Interaction

As explained in previous sections, every abstract model is a *Compound* of smaller *Parts*. What does this relation imply? What does a compound *know* about its parts? Knowledge *about* something is often called *Meta Information*.

The most obvious way to uniquely identify parts is to give them a *Name*. The concept of a human body, for example, has parts like *Heart*, *Left Arm* or *Skin*. Secondly, a compound needs to know about the *Model* of each part which may be a compound itself. But what about other knowledge like the order or position of parts within their compound?

To find an answer, the science of *Psychology* needs to be called in. It distinguishes between various aspects of a (visual) impression of the human mind, as there are *Shape*, *Depth*, *Color* or *Movement* [13]. Looking closer at these, one realizes that they are representations of the classical physical dimensions that humans use to describe the world:

- *Movement*: changing the state of something over *Time*
- *Shape*: how items would appear in a two-dimensional world, as known from *Geometry*
- *Depth* (stereo vision): adding a third dimension to shapes, so that these become three-dimensional and form a *Space*
- *Color*: not being considered a dimension, telling about how items reflect *Light*
- *Mass*: another physical value describing the world which is not considered to be a dimension

If, according to modern physics, not all of the impressions listed above are dimensions, what is common to them? – All can be used to express a special aspect of a composition relation which this paper calls *Composition Interaction*.

To the concept of an *Atom* belong a *Core* and *Electrons*. The atom provides the *Space* that the core and electrons can fill with their extension. For core and electrons, the atom represents the small universe they live in. Moreover, the atom *knows* about the *Position* (*Trajectory*) of each electron. Thus, one can say that the atom as a *Whole* interacts with its *Parts* by means of space. Electrons, on the other hand, know nothing about their own position within the atom; they do not know about the existence of the atom at all. But having a size, they indirectly exert influence on the whole atom by contributing to its overall extension in space.

A *Solar System*, as concept, has very much in common with the atom. It has a star, the *Sun*, as its core and it has *Planets* orbiting around that star. Besides the composition interaction over space that also exists here, there is another relation worth paying attention to: *Mass*. Conceptually, the solar system can be treated as a closed field of *Mass*, the sun representing the center of that mass, the planets additions. The solar system as a *Whole* knows about the masses of its *Parts*, what can be considered a conceptual interaction.

A third relation that humans use to place themselves and the environment into their very own model of the universe is *Time*. Any *Process* can be split into *Sub Processes* and such represents a structure with *hierarchical* character. In most cases, the *Order* in which sub processes are executed, is very important. Without it, no meaningful *Algorithm* could ever be created. A process knows about the *Occurrence* of its sub processes and this sequence information is stored in units of time. Moreover, the *Whole* process sets a time frame that all *Part* processes, in sum, cannot exceed. Their *Duration* is limited. Again, process and sub processes have some kind of composition relation; in this case over time.

Conceptual interactions like *Space*, *Mass* or *Time* are used by a model to position parts within its area of validity. Yet this meta knowledge is not enough. Frequently, parts have to be *constrained* to maintain the validity of the whole model. The concept of a *Table* consists of a *Top* and one to four *Legs*. The additional information herein is the *Constraint* of the number of legs to at least *one* and at most *four*.

Finally, what makes up the *Character* of an item (in the understanding of the human mind) is the *Parts* it consists of, combined with *Meta Information* about these parts. Most properties of a molecule in *Chemistry* are determined by the number and arrangement of its atoms. *Hydrogen* (H_2) becomes *Water* (H_2O) (with a totally different character) when one *Oxygen* (O) atom is added per hydrogen molecule.

Properties are based on impressions of the human mind which are often identical to what is called a *Dimension* in physics. Item *Properties* that do not result from its composed nature have to be defined additionally as size in space (expansion), in time (duration, instant), in mass (massiness) or color as speciality. While such dimension properties of an item are given as the *Difference* (size) of something, a conceptual interaction between a compound and its parts is stated as *Point* (position).

5 Cybernetics Oriented Language

The introduced *Cybernetics Oriented Language* (CYBOL) is based on the principles of *Human Thinking* as described in section 4. These principles and further concepts behind are summarized by the name *Cybernetics Oriented Programming* (CYBOP) (figure 4). They form the semantics of CYBOL. Its syntax is determined by the *Extensible Markup Language* (XML) standard and accordingly easy. It is rich enough to express models based upon the three kinds of abstraction: *Discrimination*, *Categorization* and *Composition* as well as meta information of a *Whole* about its *Parts*.

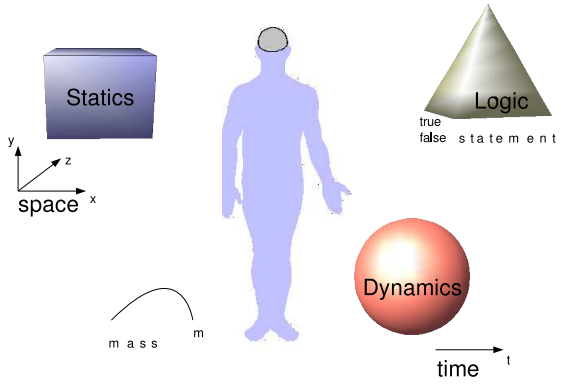

Fig. 4. CYBOP

5.1 Syntax

An XML document carries a name and can such represent a *Discrete Item*. It can also link to other documents, such as one being a *Super Category* to the item currently considered. Most importantly, XML documents have a hierarchical structure based on *Tags* which may be used to model *Parts* of a *Compound*. Tag *Attributes* keep *Meta Information* about the tag contents.

Considering these properties of XML, it seems predestinated for formally representing abstract models using the CYBOP concepts. CYBOL, finally, is XML *plus* a defined set of tags and attributes used to structure and link models meaningfully. The tags are: <model>, <super>, <part>.

5.2 Vocabulary

XML allows to define and exchange the whole vocabulary of a language. It offers two ways in which a list of legal elements can be defined: The traditional *Document Type Definition* (DTD) and the more modern *XML Schema Definition* (XSD). Besides the vocabulary, DTD and XSD define the structure of an XML document and allow to typify, constrain and validate items. The CYBOL DTD and XSD can be found at [7].

5.3 Semantics

CYBOL files can be used to model either *static* or *dynamic* aspects. In both cases, the *same* syntax (document structure) with *identical* vocabulary (tags and attributes) is applied. It is the attribute *Values* that make a difference in meaning. An *Attribute* keeps meta information about the contents of a *Tag*. In CYBOL, the tag of main interest is *part*. Its attributes contain information about its:

- Name (to identify different parts)
- Model (compound or primitive)
- Position (in space, time or mass)
- Constraint (minima, maxima and further limitations)

What is missing is a means to keep such meta information about an attribute, too. How should an interpreter know if it deals with a *compound* or a *primitive* model, with a position in *Space* or in *Time*? It is therefore necessary to *bundle* attributes in *Pairs of Two*, one attribute containing the actual value and the second attribute containing abstraction information about how the first attribute gets interpreted correctly. The only exception is the name which gets always interpreted as string of characters. The resulting attributes of the *part* tag are:

- name
- part_abstraction
- part_model
- position_abstraction
- position_model
- constraint_abstraction
- constraint_model

A list of defined, primitive abstraction values for CYBOL can be found in [7].

5.4 Example

The following example shows a minimalistic model of a (static) *Graphical User Interface* (GUI) frame.

```
<!--
    frame_example.cybol
/-->
<model>
    <part name="title"
        part_abstraction="string"
        part_model="Res Medicinae"/>
    <part name="menu_bar"
        part_abstraction="compound"
        part_model="/gui/menu_bar.cybol"
        position_abstraction="compass"
        position_model="north"/>
    <part name="status_bar"
        part_abstraction="compound"
        part_model="/gui/tool_bar.cybol"
        position_abstraction="compass"
        position_model="south"/>
</model>
```

Similar models can be built of (dynamic) workflows whereby the inputs and outputs of the part operations appear in a special order as attribute values. But this may become the topic of a follow-up paper.

6 Hardware Connection

6.1 Reflection

The use of a programming language eases model abstraction for human programmers. Special tools exist that break down models given in form of program code into their binary form, into sequences of *0* and *1*. These are called *Machine Language*, because understood by computers.

Classical programming languages have the linguistical means to express high-level *Knowledge* as well as low-level *Hardware Control Instructions*. The use of such languages inevitably leads to a mess in program code because both are mixed up. Unflexible, overly complex systems with numerous interdependencies are the result. Section 3 already criticised this weakness of traditional programming language concepts. This work makes the necessary split: Knowledge gets *separated* from hardware control.

Biological Cell Separation is one proof for this theory. The original cell forwards its configuration information in form of a *Desoxy Ribo Nucleic Acid* (DNA) to the new cell. The new cell uses this knowlege to create new organelles and to function correctly. Each cell represents a system with different *Hardware* that it controls but all cells (in one-and-the-same biological creature) use the same configuration.

Regions of the Human Brain are another example. The main knowledge is stored in the *Cerebral Cortex*. Other regions more or less just control the exchange of knowledge through input/ output organs (hardware), for communication with other biological systems.

6.2 Cybernetics Oriented Interpreter

The CYBOL language described in section 5 is just another form of storing knowledge. It can therefore also be called a *Knowledge Modelling Language*.

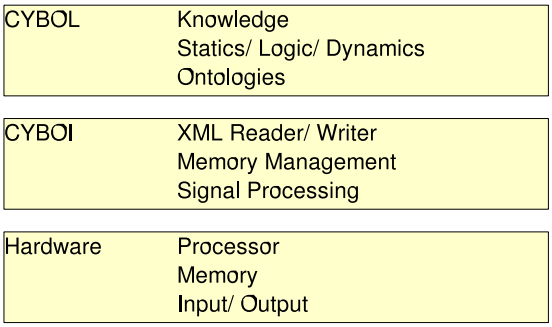

Fig. 5. CYBOI as Interface between CYBOL and Hardware

When CYBOL files contain the knowledge that defines a system, a counterpart is needed to execute that system on computer hardware. The *Cybernetics Oriented Interpreter* (CYBOI) is able to handle this task (figure 5). CYBOI is written in the C programming language and currently supports the *Linux Operating System* (OS) only. It represents, so to say, the interface between operating instructions of the computer hardware and system models defined in CYBOL.

CYBOI is responsible for managing any kind of hardware communication, that is input, output, memory access and processor instruction calls. CYBOI Signals can be assigned priorities, a language (protocol) to communicate with other systems and they are processed by one single loop (figure 6). Also, there is only one single container structure which CYBOI uses to dynamically store knowledge. It such avoids the known problems with container inheritance [6]. Following *Euclidian Geometry*, multi-dimensional *Models* consist of maps; two-dimensional *Maps* consist of arrays; one-dimensional *Arrays* represent and manage an area in the computer memory.

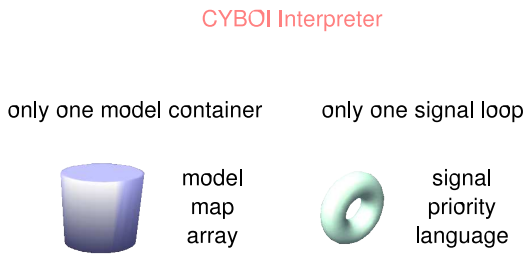

Fig. 6. CYBOI Model Container and Signal Loop

The more hardware driving functionality CYBOI implements, the more it develops towards an operating system – with one difference to current OS: It is free of any configuration information but knows how to *handle* knowledge.

7 Related Work

There are a number of efforts that go into a similar direction like CYBOP [7]. Basically, every application that stores configuration data (colours, fonts) does use some kind of knowledge model for the file or database to save in. However, they all are limited to their corresponding field. What this paper proposes is, in short, to store complete systems in special configuration files in CYBOL format. CYBOP wants to show an overall approach and provide the means (CYBOL/ CYBOI) to build abstract software models for any possible application layer, may it be a domain, user interface, workflow, data transfer object or storage.

The two projects mentioned following do related work, with focus on the medical domain.

Open Infrastructure for Outcomes (OIO) [8] is a Web-based data management system that uses forms (and workflows) which are defined in XML. Its most critical point is that OIO forms mix user interface with domain model data. Moreover, it misses a clear theory behind and does not distinguish static and dynamic models.

Open Electronic Health Record (OpenEHR) [9] is a standardization effort that arose from a European initiative. Its *Dual Model Approach* also influenced CYBOP. The project's main aim is the creation of knowledge templates (which it calls *Archetypes*), for which an own *Archetype Definition Language* (ADL) was defined. A lot of emphasis is placed on constraint inclusion, to ensure correct models. However, OpenEHR's model concepts are not based on abstraction as it happens in the human mind. They do not clearly distinguish between constraints, positions and the actual model information. There is no facility for translating between archetypes [2]. It offers only static archetype models, no dynamic workflows. ADL seems overly complex and difficult to understand. The whole project still lacks implementation experiences and practical proof of workability.

8 Summary

This paper means that wild *Dependencies* are a major reason for error-prone, unstable, unflexible, unmaintainable software systems. Two facts causing such dependencies are the *Bundling* of static and dynamic properties by object-oriented languages and the *Mix* of knowledge and hardware control in traditional programming languages. This information mix additionally forces software development projects to run through a course of different abstraction steps which would not differ if one common knowledge abstraction were used.

As solution to the above's problems, this document suggests to build software systems after the concepts of *Human Thinking*. The approach, named CYBOP, such follows the recommendations of the science of *Cybernetics* and its specialization *Bionics*, whereby biological principles should be applied to the study and design of engineering systems. An abstract model as formed in the human mind represents an *Item*, *Category* and *Compound*, at the same time. Additionally, it contains *Meta Information* about its parts. This information often corresponds to physical dimensions and determines whether the model is an abstraction of *static* or *dynamic* real-world aspects.

The introduced *CYBOL* language has the semantics to express knowledge models as used by human thinking. It allows to create complete application systems. Its syntax is based on *XML* which results in absolutely platform- independent system definitions. CYBOL files get interpreted by the *CYBOI* interpreter and can be changed at runtime. CYBOI manages all hardware access. It concentrates model instances and signal handling in one place and such avoids memory leaks and endless loops.

CYBOL models could be displayed graphically, using special design tools. But their *formal definition* also allows them to be used as main abstraction throughout all phases in a software project's lifetime. Analysts and experts can start their work by creating rudimentary CYBOL models (defining static structures and dynamic processes) which software designers can later complete and check for correctness. The implementation phase becomes superfluous at all: CYBOL models already represent the system to be built, no further code is needed! It is hard to imagine the amount of saved time and costs for software projects. Even better: Experts are placed in a position to, themselves, actively help creating systems.

References

1. Remigiusz Bierzanek and Janusz Symonides. *Prawo Miedzynarodowe Publiczne (citing S. E. Nahlik)*. Wydawnictwa Prawnicze (PWN), Warszawa, 5 edition, 1999. http://www.wp-pwn.com.pl.
2. Minoru Development. Open health mailing list, 1999-2004. ¡openhealth-list@minoru-development.com¿.
3. Christian Heller, Jens Bohl, Torsten Kunze, and Ilka Philippow. A flexible software architecture for presentation layers demonstrated on medical documentation with episodes and inclusion of topological report. *Journal of Free and Open Source Medical Computing (JOSMC)*, 1(26.06.2003):Article 1, June 2003. http://www.josmc.net.
4. Christian Heller, Torsten Kunze, Jens Bohl, and Ilka Philippow. A new concept for system communication. *Ontology Workshop at OOPSLA Conference*, October 2003. http://swt-www.informatik.uni-hamburg.de/conferences/oopsla2003-workshop-position-papers.html.
5. Ludwig Jaeger. Linguistik: Ohne sprache undenkbar. In *Gehirn & Geist*, volume 2, pages 36–42. Spektrum der Wissenschaft, http://www.spektrum.de, 2003. http://www.gehirn-und-geist.de.
6. Peter Norvig. The java iaq: Infrequently answered questions. http://www.norvig.com/java-iaq.html.
7. CYBOP Project. Cybernetics oriented programming (cybop), 2002-2004. http://www.cybop.net.
8. OIO Project. Open infrastructure for outcomes (oio), 2000-2004. http://www.txoutcome.org.
9. OpenEHR Project. Open electronic health record (openehr), formerly good electronic/ european health record (gehr), 2000-2004. http://www.openehr.org.
10. Peter Ripota. Das universum hat ein bewusstsein! In *P.M. Magazin*, pages 21–25. Hans-Hermann Sprado, September 2003. http://www.pm-magazin.de.
11. Christoph Schoenhofer. Unternehmensberatung: Die neuromanager. In *Gehirn & Geist*, volume 2, pages 74–75. Spektrum der Wissenschaft, http://www.spektrum.de, 2003. http://www.gehirn-und-geist.de.
12. John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole, Thomson Learning, Pacific Grove, 1997.
13. P. Stoerig. Hirnforschung – visuelle wahrnehmung: Blindsehen. In *Gehirn & Geist*, volume 2, pages 76–80. Spektrum der Wissenschaft, http://www.spektrum.de, 2003. http://www.gehirn-und-geist.de.