We are IntechOpen,
the world's leading publisher of
Open Access books
Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

154
Countries delivered to

Our authors are among the
TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

1

Machine Biological Clock: Exploring the Time Dimension in an Organic-Based Operating System

Mauro Marcelo Mattos
FURB – University of Blumenau
Brazil

1. Introduction

Ubiquitous Computing (UbiCom), Autonomic Computing (AC) and Organic Computing (OC) research has produced a substantial body of work dealing with smart devices, smart environments and smart interaction technologies.

Ubiquitous computing was introduced by (Weiser, 1991) and is related to a vision of people and environments augmented with computational resources providing information and services when and where they could be desired, going beyond than just infrastructure aspects, and suggesting new paradigms of interaction inspired by widespread access to information and computational capabilities (Abowd & Mynatt, 2000) (Poslad, 2009). This vision involves social, technological, engineering and foundational questions (Milner, 2006).

UbiCom environments are increasingly challenging domains when compared with those traditional – also not so easy to deal with traditional computing applications domains. According to (Brachman, 2002) in such scenario there exists the need for a software infrastructure that supports all sorts of heterogeneities (hardware, operating systems, networks, protocols and applications).

Autonomic Computing is related to someone or something acting or occurring involuntarily. It is related to the ability to manage the computing enterprise through hardware and software that automatically and dynamically responds to the requirements of the business. This means self-healing, self-configuring, self-optimizing, and self-protecting hardware and software that behaves in accordance to defined service levels and policies (Murch, 2004)(Balasubramaniam, et al., 2005).

Organic Computing is a research field emerging around the conviction that problems of organization in complex systems in computer science, telecommunications, neurobiology, molecular biology, ethology, and possibly even sociology can be tackled scientifically in a unified way, by means of which progress in understanding aspects of organization in either field can be fruitful in the others (Würtz, 2008). OC systems are based on a general architecture, which would permit users to create specific applications by defining goal hierarchies (Malsburg, 2008) taking advantages of one of the key attributes of biological systems making it possible to adapt and change on multiple time scales as they evolve, develop, and grow, and they should do so without external direction or control (Bellman, Landauer, & Nelson, 2008).

The pervasiveness characteristic of these demands also implies the growing dependency on the expectance to obtaining the proper services when the system is fault-free and especially when it encounters perturbations. So, it is important to qualitatively and quantitatively associate some measures of trust in the system's ability to actually deliver the desired services in the presence of faults.

Since the first steps in the computing history we have seen the field of Software Engineering expand in several ways including the application of software architecture principles to the development of systems. Software architecture involves both the structure and organization by which modern system components and subsystems interact to form systems, and the properties of systems that can best be designed and analyzed at the system level. The importance of software architecture for software development is widely recognized, yet transfer of innovative techniques and methods from research to practice is slow (Krutchen, 2004) (Osterweil, 2007)(Kruchten, Capilla, & Dueñas, 2009)(Buschmann, 2010) and costly (Lagerström, von Würtemberg, Holm, & Luczak, 2010) due to rapid and continuous technology changes.

One important aspect to be pointed is that the current computing platform is made upon a vast collection of code – operating systems[1], programming languages, compilers, libraries, run-time systems, middleware – and hardware that make possible a program to execute. This platform has not evolved beyond computer architectures, operating systems (OS), and programming languages of the 1960's and 1970's (Hunt G., et al., 2005)(Hunt & Larus, 2007). In consequence, application and operating system errors are a continuing source of problems in computing. Existing approaches to software development have proven inadequate in offering a good tradeoff between the assurance, reliability, availability, and performance in such a way that software remains notoriously buggy and crash-prone (Naur & Randell, 1969) (Anderson, 1972) (Randell, 1979) (Linde, 1975)(Kupsch & Miller, 2009),(Ackermann, 2010). In this context, the OS is probably the most crucial piece of software that runs on any computer (Iyoengar, Sachdev, & Raja, 2007).

The preceding paragraphs bring us a scenario that is contrasting: from one side the landscapes of software engineering domains are constantly evolving and for the other side, the computing environments (hardware, OS, telecommunication infrastructure and tools) have historically proved not be robust enough. In this ever-changing scenario, the mainstream research in software engineering goes in a direction trying to propose innovative solutions in the realm of building, running, and managing software systems.

In order to find an appropriate solution to development and design of the new class of systems an appropriate paradigm seems necessary. We choose to take the opposite direction towards the past to try to figure out what could be changed in the beginning of the process in order to minimize the recurrent problems that we are faced in developing and using software. As a consequence we proposed a new software architecture where:

- the only alive (runnable) entity is the operating system, and
- the operating system has the ability of learning based on past experiences on what to do, how to do it and when start learning about solving tasks.

[1] In this work, we refer to the concept of *operating system* in a broader sense, involving the categories of general purpose, embedded, stand-alone or networked, because we need to get an overview first, before examining each class in depth. Moreover, the aspects under review do not require differentiation between these classes.

In the present work, we aim at attracting the reader's attention towards the conception of a system with the ability of knowing how to perform tasks and how to self-adapt to the fluctuations of resource availability when interacting with the surrounding environment. We call this system as a Knowledge-based Operating System (KBOS).

The work is organized as follows: a problem's contextualization related to the current paradigm of computing systems development is presented in section 2; some fundamental concepts are reviewed in section 3; a knowledge-based operating system concept in section 4; section 5 presents some related works and in the conclusion section the final comments are presented.

2. Current paradigm

Sequential programs can be described by a single flow of execution and by the use of simple programming structures such as loops and nested function calls. The execution context of these programs in some point of the run time is defined by the value of the program counter, the value of the cpu registers and the content of the program's stack.

The figure 1 presents an overview of the current paradigm in computing. From a software development perspective, to develop software is to follow some method (software development life cycle) in order to go from requisites analysis to implementation. Also, let us to consider that a program can be represented by a development team (figure 1b) and that a particular software development team, in general, does know nothing about other team's work. This could lead us to situations like:

- similar code continues to be developed by different teams;
- programming errors continues to be introduced in different points of the development steps;
- information about the final run-time environment remains unavailable for the OS;
- race conditions between non synchronized programs remains leading to instabilities;

In other words: the development team does not have ENOUGH information about ALL POSSIBLE ENVIRONMENTS where the software will be used[2].

From the users perspective, to use a software is a matter of clicking over some icon and expecting the corresponding program to start running. The user knows about the purpose of a program and has some expectation about its behavior.

From the operating system's perspective, all knowledge that is previously known is about slicing (and possibly trying to protect) binary (executable) code over the time (figure 1a).

Regardless of which method was chosen to develop a particular application, at some point we will move to the phase of code generation. In this moment, all the documents (and source code) will be stored in files (figure 1b) and the compiler will generate a string of bits that we used to call: a program.

At this moment, the OS comes to scene - remembering that the main purpose of an operating system is to share computational resources among competing users. To do this efficiently a designer must respect the technological limitations of these resources (Peng, Li, & Mili, 2007).

One of the difficulties of OS design is the highly unpredictable nature of the demands made upon them mainly because the relationship between different applications are not considered as a functional/non-functional requisite at the design time. This happens

[2] We should to consider that each user's machine probably will have different hardware and software configurations that in some moment could be running a particular buggy combination of factors.

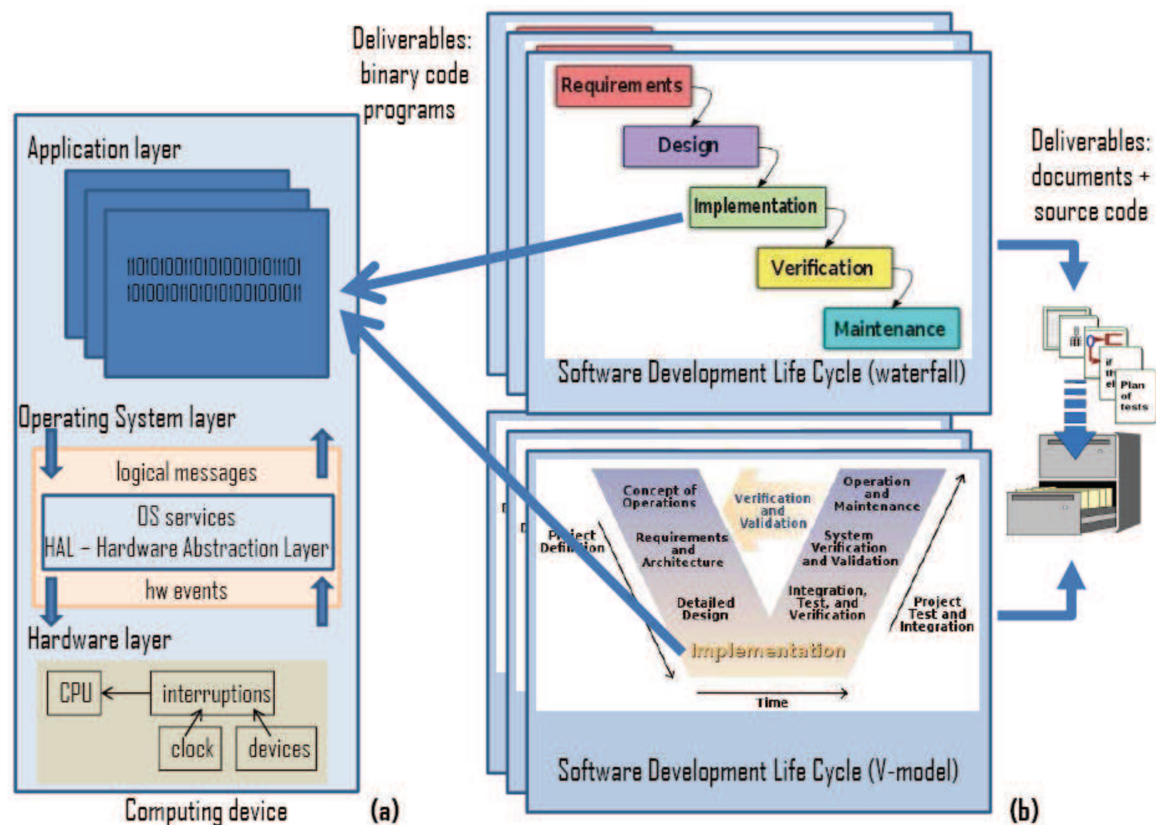


Fig. 1. Current paradigm: software from an inside-out perspective (a), and from an outside-in perspective (b).

because the structure of an OS requires a series of fine-grained event-handler functions for handling events. These event-handler functions must execute quickly and always return to the main event-loop.

Behind the "software layer that manages the hardware" concept (Tanenbaum, 2008), an OS could be better described as a software architecture (Perry & Wolf, 1992) which embeds a large number of design decisions related to hardware interface, programming languages and tools that have a direct impact in almost every software that will be deployed. So, an OS architecture involves a set of functional components related to management of processes, memories, files, devices (input/output operations), security and user interface. In general they are organized in layers.

The first level is the hardware that requires the OS attention by emitting signals to the CPU thru some kind of interrupt model. These hardware events are converted to some kind of logical messages to be dispatched to the application running on that computing device (figure 1a). This conversion exports an abstract view on hardware so that programmers do not have to deal with low level details.

The analysis on the set of clues presented leads us to speculate on the influence that some of the key concepts related to current paradigm: multiprogramming[3], operator, and program - can contribute to the recurrence of the historical problems[4].

[3] There is a class of embedded applications that, for the very specific nature, are not affected by the principle of interruption (have deterministic behavior). Even these, however, could be to some extent, included in this reflection.

2.1 Multiprogramming

The computing device, in general, can run several applications at a time leading us to some kind of multiprogramming environment. The main difficulty of multiprogramming is that the concurrent activities can interact in a time-dependent manner which makes it practically impossible to locate programming errors by systematic testing.
Perhaps, more than anything else, this explains the difficulty of making operating systems reliable (Hansen, 1973), (Hansen, 1977) (Post & Kagan, 2003).
One of the most fundamental design decisions in conceiving a new OS architecture is related to the definition of the type of kernel: non-preemptive or preemptive. This decision imposes a tradeoff between the coupling in the time domain and resource sharing. A non-preemptive kernel makes the OS able to share resources among tasks but couples the tasks in the time domain while a preemptive kernel decouples the tasks in the time domain but enforces the resource sharing (Samek, 2009).

2.2 Operator

Another interesting aspect to be considered is about the figure of a computer operator[5].
In 1961 Klausman wrote: "… I define a *computer operator* as a job responsibility of a person who is in charge of the computing equipment while it is in normal operating condition. The equipment includes the processor, its console or supervisory control panel, and the peripheral equipments on-line or off-line. The operator may have assistants to change tapes, paper forms or the like. Normal operating condition is that in which the system is able to operate in continuous or automatic mode without intervention for relatively long periods of time. These periods may be interrupted by occasional transient errors which do not cause maintenance service. The operator's responsibilities include the running of production programs, programs being *debugged*, and service routines, such as compilers, tape correction routines, etc. The operator's responsibility also includes the diagnosis and action taken as a consequence of transient errors. In addition is the general area of communications into which the operator fits. To intelligently operate the system his knowledge should transcend mere ability to push buttons, an activity which may steadily decrease with the growth in sophistication of the programming art and engineering developments during the sixties… It is conceivable that a data processing system will be completely automatic. A real time clock built into the system will turn it on in the morning or the middle of the night. Automatic tape changes will mount and dismount tapes - feed cards, forms and the like. And the operator - where is he? He isn't - the function ceases to exist. This may not happen tomorrow or next year but it is coming. In an industry which is literally begging for competent personnel it seems to me that operators have nothing to fear from this progress, for more challenging jobs have appeared and will continue to be created for decades to come…" (Klausman, 1961).
Currently, the figure of the computer operator was replaced by the figure of the end user and, in this context, one of the programmer's role is to arrange virtual buttons in a graphical user interface for user to press them and thus make the program work.

---

[4] It is important to emphasize that we are analyzing some characteristics in order to understand what could be the cause of recurring issues, and we are not arguing for or against the current paradigm.

[5] Even considering the class of embedded operating systems for the specific purpose, the concept of operator remains valid (albeit virtually) once the set of interactions between the external world and the controlled device is performed through a set of well established interfaces – replacing buttons for function calls. Under the functional perspective, there is no difference between pressing a button or calling a function.

2.3 Program

One last aspect to be discussed refers to the concept of program (Haigh, 2002): basically a program is a binary expression of some algorithm written in a programming language.
The classical computing model is based on detailed algorithmic control, rests entirely on the insight of the programmer into the specific application of the program and has a strong dependency of abstraction layers. The machine is deterministic and blindingly fast, but is considered as totally clueless. The programmer is in possession of all creative infrastructures, in the form of goals, methods, interpretation, world knowledge and diagnostic ability (Malsburg, 2008).
This approach can work for any well-defined and sufficiently narrow tasks. But, if the system fails, the programmers would diagnose and debug the errors. They would determine what knowledge to add or modify, how to program it, and how to modify and rebalance the pre-existing programs to accommodate the new performance without harming the parts that already worked well (Hayes-Roth, 2006).
Automation in adaptation, learning, and knowledge acquisition is very limited – a tiny fraction of the overall knowledge required, which the engineers mostly prepared manually. The strategy to cope with the increasing complexity of software systems is to adopt some kind of infrastructure based on several levels of abstractions (Kramer & Magee, 2007), (da Costa, Yamin, & Geyer, 2008).

3. Fundamental concepts

"…Computers, unfortunately, are not as adept at forming internal representations of the world. ... Instead of gathering knowledge for themselves, computers must rely on human beings to place knowledge directly into their memories…" (Arnold & Bowie, 1985).
Before proceeding, we must establish a conceptual basis related to the context of this work.

3.1 Data, information, knowledge, knowledge-acquisition

To (Frost, 1986): *Knowledge* is the symbolic representation of aspects of some named universe of discourse, and *Data* is a special case of knowledge and means the symbolic representation of simple aspects of some named universe of discourse.
(Meadow & Yuan, 1997) in they work on measuring the impact of information on development affirm that "we can consider that the terms *data*, *information* and *knowledge* represent regions in an epistemological continuum. They are not specific points, because each one has many definitions and variations. *Data* generally means a set of symbols with little or no meaning to a recipient, *information* is a set of symbols that have meaning or significance to its recipient, and *knowledge* means the accumulation and integration of information received and processed by a recipient".
Although there is no unanimity (Lenat & Feigenbaum, 1988) (Davis, Shrobe, & Szolovits, 1993) (Duch, 2007), the researchers agree that *knowledge representation* is the study of how knowledge about the world can be represented and what kinds of reasoning can be done with that knowledge.
Knowledge in the context of this work is conceived as being a set of logical-algebraic operational structures that makes possible to organize the system's functioning according to interconnection and behavior laws.

It is well known that a significant obstacle to the construction of knowledge-based systems is the process of *knowledge acquisition*(Shadbolt, O'hara, & Crow, 1999). The key to this process is how we may effectively acquire the knowledge that will be implemented in the knowledge base. In an operating system environment, this is not an easy task. It is usually done by hooking the calls to operating system application programming interface (API) and recording logs for further analysis (Skjellum, et al., 2001). This approach is a time and resources consuming process and presents, as the main drawbacks:

i. the data gathering process impacts the overall performance, influencing other applications that aren't involved in the application context being considered;
ii. this impact on performance also interferes with the application being considered;
iii. and this scenario probably will be different from that of where the application was developed.

3.2 Intelligence, machine intelligence and finite state machines

Also, there is no consensus on the definition of intelligence. (Legg & Hutter, 2007) in their work on "machine intelligence" states that, in general, most definitions share the fact that *intelligence* is a property of an entity (an *agent*) which interacts with an external problem or *situation* (usually unknown or partially known), and has the ability to succeed with respect to one or more goals (the *goals*) from a wide range of possibilities (not just some specific situations).

A particular view for machine intelligence is presented by (Costa, 1993) where he introduces a definition for the concept of machine intelligence, shows the practical possibility to this definition and provides an indication of its need, it gives you an objective content and shows the value and usefulness that such a definition may have to the computing science in general, and artificial intelligence in particular. Rocha Costa started from the intelligence definition given by J. Piaget and established how the conditions for such a definition could be interpreted in the machine domain. The definition presented assumes that it must be recognized the *operating autonomy* of the machines. This leads to abandon, or at least put on second plan, the perspective of contrived imitation for intelligent behavior from humans or animals and adopt the point of view that he calls *naturalism* - to consider machine intelligence as a natural phenomenon on the machines.

A common and straight way of modeling behavior is extending the event-action paradigm to explicitly include the dependency on the execution context through a finite state machine (FSM). An FSM is an efficient way to specify constraints of the overall behavior of a particular system. Also FSMs have an expressive graphical representation in the form of state diagrams - directed graphs in which nodes denote states, and connectors denote state transitions. The FSM has a drawback, the phenomenon known as state explosion, related to the fact that there is an implicit notion of repetition of states. To make its use more practical, state machines can be supplemented with variables. In this case, they are called extended state machines, and can apply the underlying formalism to much more complex problems than could be practical without including the variables (Samek, 2009).

3.3 Time and cognition

"…I'm trying to understand how time works. And that's a huge question that has lots of different aspects to it. A lot of them go back to Einstein and space-time and how we measure

time using clocks. But the particular aspect of time that I'm interested in is *the arrow of time*: the fact that the past is different from the future. We remember the past, but we don't remember the future. There are irreversible processes. There are things that happen, like you turn an egg into an omelette, but you can't turn an omelette into an egg. …"(Biba, 2010)

Despite the importance, of the concept of time has been discussed in several venues (Church, 2006), (Stenger, 2001). However, it is undeniable that THE CONCEPT is implicitly linked to daily activities by establishing a sequence, seemingly logical, of real-world events.

According to (Carroll, 2008), from the perspective of physics, "the nature of time is intimately connected with the problem of quantum gravity. At the classical level, Einstein's general relativity removes time from its absolute Newtonian moorings, but it continues to play an unambiguous role; time is a coordinate on four-dimensional space-time, however, it measures the space-time interval traversed by objects moving slower than light.

Under the Quantum Mechanics perspective, there are considered some fundamental aspects like the position and momentum of a particle what imperfectly reflect the reality of the underlying quantum state. It is therefore perfectly natural to imagine that, in a full theory of quantized gravity, the space-time itself would emerge as an approximation to something deeper. And if space-time is an emergent phenomenon, surely time must be".

Once the knowledge representation is captured, inferences can be made including extending forward from the known past and present to the unknown (prediction or statistical syllogism) and/or determining the causality by extending from the known data back to hypothesis (explanation or abduction) (Josephson & Josephson, 1994).

Thus, every knowledge representation model requires a representation of time, of the temporal relationship between events and has to deal with uncertainty. In some systems, the time model is such that the actions should be considered instantaneous, and only one action can occur at some given time, while in others, where there is an association between an action and a time reference, the inference module can automatically derive other relations.

In a more philosophical perspective (Overton, 1994) states that the cycle of time is a deep metaphor entailing a relational field of both nonclosed cycles (spirals) and direction that emerges in a broader sense across a several scientific disciplines. In the context of the organic narrative, the cognition and personality are understood as emerging from a fundamental relational theory of the embodied mind. In the context of the mechanical narrative, the development is understood as being limited to variation (and only variation), and cognition and personality emerge from a theory of the computational mind.

In computing, a more practical approach on this subject has been addressed in research on intelligent agents. To illustrate, we selected two works in which the relationship between time and are intrinsic cognition although greater attention is devoted to the cognitive aspect. A promising approach called *action awareness* is based on to provide agents with reflective capabilities where agents can reflect on the effects and expected performance of their actions (Stulp & Beetz, 2006). Another approach is based on an *efficient thought* concept (Hayes-Roth, 2006) that is based on a list of eight steps that the most complex organizations, in general, perform in parallel. This approach states that the intelligent being:

- observes what's happening in the environment,
- assesses the situation for significant threats and opportunities,
- determines what changes would be desirable,

- generates possible plans to operate those changes,
- projects the likely outcomes of those plans,
- selects the best plan, and
- communicates that plan to key parties before implementing it.

Throughout the process, the intelligent being validates and improves its model.

3.4 Biological clock

According to (Schmidt, Collette, Cajochen, & Peigneux, 2007) "… There is evidence that the interaction between homeostatic and circadian factors is not linear throughout the day and can affect a wide range of neuro behavioral events. However, the impact of potential time-of-day variations on brain activity and cognitive performance remains largely ignored in cognitive psychology and neuropsychology, despite the fact that Ebbinghaus (1885/1964) already reported more than one century ago that learning of nonsense syllables is better in the morning than in the evening...".

According to (GSLC, 2010), "living organisms evolved an internal biological clock, called the circadian rhythm, to help their bodies adapt themselves to the daily cycle of day and night (light and dark) as the Earth rotates every 24 hours. The term 'circadian' comes from the Latin words for about (*circa*) a day (*diem*). Circadian rhythms are controlled by *clock genes* that carry the genetic instructions to produce proteins. The levels of these proteins rise and fall in rhythmic patterns. These oscillating biochemical signals control various functions, including when we sleep and rest, and when we are awake and active. Circadian rhythms also control body temperature, heart activity, hormone secretion, blood pressure, oxygen consumption, metabolism and many other functions. A biological clock has three parts: a way to receive light, temperature or other input from the environment to set the clock; the clock itself, which is a chemical timekeeping mechanism; and genes that help the clock control the activity of other genes".

People (and other animals) are able to perceive the duration of intervals between events however the organism's internal clocks are not exactly 24 hours long. Associative learning is dependent upon time perception, and the mechanisms of time perception are related to an internal clock. In situations in which there are many different time intervals, these can be combined for the assessment of the typical interval (Schmidt, Collette, Cajochen, & Peigneux, 2007).

3.5. Situated agents

"...unfortunately, programming situated agents is quite difficult. Interacting with a dynamic and largely unpredictable environment introduces a number of significant problems. Most of these problems are related to the way the agents use the plans that determine their behavior. Traditionally, plans were used literally; the agent did exactly what the plan said. This placed a heavy burden on the plan maker, because it had to foresee all the possible ways in which the agent's interaction with the environment might unfold. Today, it becomes clear that an agent should have the ability to interpret plans in a more sensible and context-dependent way; it should be able to improvise, to interrupt, resume and sequence activities, to actively forage for information and to use the current situations to disambiguate references in its plans" (Schaad, 1998).

We find that statement important to resume the actual paradigm. In our point of view, the actual paradigm can be introduced as follows: “unfortunately, programming is quite difficult. Interacting with a dynamic and largely unpredictable environment introduces a number of significant problems. Most of these problems are related to the way the programmers develop programs that determine their behavior. Traditionally, programs are used literally; the program does exactly what was programmed to do. This place a heavy burden on the programmer, because he has to foresee all the possible ways in which the program’s interaction with the environment might unfold.
Unfortunately, our programs, in general, continues to be forged as static pieces of instructions.

3.6 World model

One aspect of fundamental importance in the robotics research area, and one that it is neglected by the operating systems designers refers to the fact that in robotics projects there is always a mapping function between reality and an internal representation denominated "world model". In other words, there is some form of explicit environment representation where the robot will operate. And it is this world model that determines what decisions are made.
It is important to make a distinction between two types of world models:
i. those that only describe the current state of the agent’s surroundings, and
ii. those that include more general knowledge about other possible states and ways of achieving these states. The first models are commonly referred as environment models and typically include some kind of spatial 3-D description of the physical objects in the environment. It contains dynamic and situation-dependent knowledge and can be used, for instance, in navigation tasks. The models of the second kind are referred as world models, and typically include more stable and general knowledge about: objects, properties of objects, relationships between objects, events, processes, and so on (Davidsson, 1994).
Accordingly Grimm et. al (2001), the main requirements to be reached in this class of projects are: robustness; reliability; modularity; flexibility; adaptability; integration of multiple sensors; resolution of multiple objectives; global reasoning, and intelligent behavior.
This aspect is also considered in autonomous agents research area. As stated in (Franklin & Graesser, 1996) “... Autonomous agent means a system situated in, and part of, an environment, which senses that environment and acts on it, over time, in pursuit of its own agenda. It acts in such a way as to possibly influence what it senses at a later time...”.
That is, the agent is structurally coupled to its environment. If an operating system does not have an internal representation of its own relationship with surroundings, how can we suppose that it can make intelligent decisions? That is one of the main problems to be solved by the designers of new generation operating systems. It must be clear that by affirming that the operating system doesn't have an internal representation of its internal state we mean that it's not enough to collect statistical data about all the processes and other countable things that occur when the system is running. Instead, it has to collect them in order to be able to infer something about what is happening at some particular moment. This is a much more complex process that cannot be achieved by writing multitudes of scripts and building lots of administration tools.

3.7 Comments

In this section, we presented a set of concepts for which there is no unanimity among researchers. It was not our intention to present a complete review of the disciplines, but point out a few aspects that we consider important in the context of this work.
We followed a path starting from the more abstract concepts (knowledge, intelligence, time) towards the more concrete ones (biological clock).
We are considering the biological clock as the starting point to establish a time unit compatible with that found in humans and over which we do our daily tasks (including learning, planning and dealing with uncertainties - requisites from Ubicom, AC, and OC) and that does have no relationship with the real time clock used in machines.
After these considerations we can conclude that the complex and dynamic nature of the environment were software solutions are developed (and where they will be executed) has the effect that the operating system:

- does not have complete control over the environment;
- does not have the capacity to devise complete models of its environment (not only its counters and pointers);
- does not possess complete information about the environment, and
- cannot completely trust the information it does have, because it is usually uncertain, imprecise, noisy, or outdated due to the nature of its perceptual processes.

At this point we have collected evidences pointing to the expectation that we need to build systems capable to export some kind of intelligent behavior. To achieve these goal artificial systems must have direct access to their environments beyond the information stored in logs It is not enough to have elaborated reasoning, learning and planning capabilities because such an intelligent entity has to be able to autonomously acquire its required information through perception and carry out contemplated actions. In other words, it is necessary to make those "intelligent entities" more sensitive to context, enabling them to sense their environment, decide which aspects of a situation are really important, and infer the user's intention from concrete actions. Those actions may be dependent on time, place end/or even the past interactions with user.
These limitations have been a central driving force behind the creation of a new operating system based on knowledge abstraction. The main goal is to bring together knowledge about artificial intelligence, robotics and physics in order to produce a new class of operating systems able to cope with the presented challenges.

4. A Knowledge-based operating system model

The novel concept introduced in Mattos (2003) says that a knowledge-based operating system (KBOS) is: "an embodied, situated, adaptive and autonomic system based on knowledge abstraction which has identity and intelligent behavior when executed". The whole system is built inside a shell which gives the endogenous characteristic. A hyper dimensional world model enables the entire system to perceive evolving and/or fluctuating execution conditions.
The endogeneity characteristic of the system insofar as the world model is surrounded by the hardware, i.e. the world model *is* the system. The world model can be characterized as the surrounding membrane of a biological cell. The nucleus is the hardware. Therefore, the membrane acts as an interface between the external environment and internal environment. Using the analogy of the cell, we cannot break through the membrane to access the inner

parts of it. So any form of influence in the cell must occur in a process similar to osmosis, i.e. provide stimulus to the interface which will translate the stimulus to the internal representation of the cell.

4.1 A KBOS World Model

We have identified 3 dimensions over which such a new operating system paradigm has to be based:

- physical dimension,
- behavioral dimension, and
- temporal dimension

The physical dimension describes the physical hardware components and their structural relationship. The behavioral dimension is described by extended state machines. Each device has a state machine for describing is physical primary behavior we called this as: *physical context of a device* (PCD). A state machine describes the dynamic aspects of the component's behavior. The current state of some device is represented by a string of bits (figure 2).

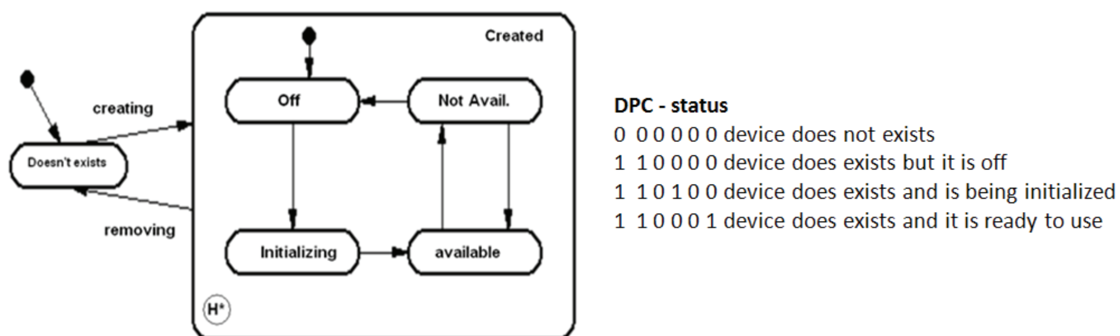


Fig. 2. Physical context of a device

Each device has a state machine that describes, in a more high level of abstraction, the functional aspects of it - we call it a *logical context of a device* (LCD) (figure 3).

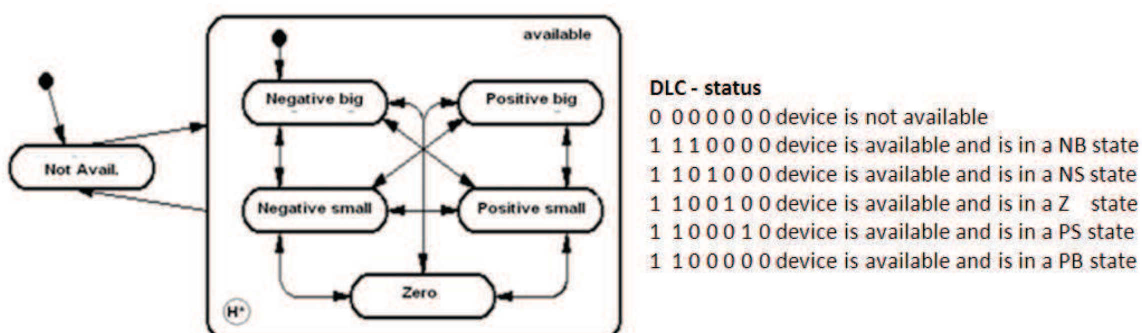


Fig. 3. Logical context of a device

One aspect that we would like to point is that the LCD use a fuzzy notation to express functional aspects of a particular device like available space (i.e. a disk unit could express space availability ranging from completely full to completely empty), communication link availability and so on.

Merging all the PCD and LCD results in a bit mask that represents the current state of the world (figure 4). This *world status word* (WSW) is used to trigger the execution of plans that were conceived as context-sensitive. It can be observed that the status word may also to represent some world's configurations for which yet there are no plans available.
For example, we could have an action plan describing what TO do in a situation where the network connection is good and the disk space is at least 50% available - let's call it the ideal solution. In a situation where the network connection is bad and disk space is less than 20% could lead to bad behavior in the plan. Perceiving this, a KBOS can start the learning stage, where it will test whether the optimal solution will work well or it will fail at some point of the present situation. If the solution works well, the system learns and registers at its knowledge base that the ideal solution also works for this situation. If not, the system will take to choose alternative plans in order to cope with the situation.
One could to observe that the bit mask is highly sensitive to fluctuations of the possible states of the world what can lead to a combinatorial explosion of states. Thus, a requirement for development of applications for a KBOS is explicitly to conceive exception conditions for each individual application. This characteristic leads to the development of more context-sensitive applications.
The description above characterizes one difference in designing software when compared with the traditional way of doing (figure 1). In our approach, the software development process should be guided by the dynamics of the application. Furthermore, we believe it should be abolished the phase of binary code generation - at the end of the compiling process as we always have done until now.
In this new perspective, the process of building a program should to finish by delivering a set of technical information to be provided to the assimilation interface of a KBOS which is the module effectively responsible for to transform that information into execution plans.
The act of transforming a program in an execution plan for a KBOS is a three-step process.
The first step involves the traditional process of software development (conception, design, implementation, testing) - including the constraints demanded by the context of a KBOS environment.
The second step involves generating, instead of an executable code, a meta-model containing:

- an extended state machine describing the dynamic behavior of the entire application and,
- the source code associated with the effective implementation of the application logic for each state/sub-state.

The third step involves submitting the meta-model to the KBOS assimilation interface for effective generation of a set of execution plans.
Our contribution stems from the fact that we are recognizing the importance of providing for the KBOS more than binary code (executable) to manage. Figure 5 reinforces the fact that we are not arguing about how we use to obtain information about the domain of a particular application, or how we should map this information in terms of software architecture or even trying to change the current paradigm of programming.
An execution plan is built in a parallel functional decision tree (Schaad, 1998) format and represents the lowest level of code that KBOS recognizes and executes. So, in a broader sense, of knowledge of a KBOS emerges from a library of execution plans and from the system's experience in to execute them according to environment fluctuations.

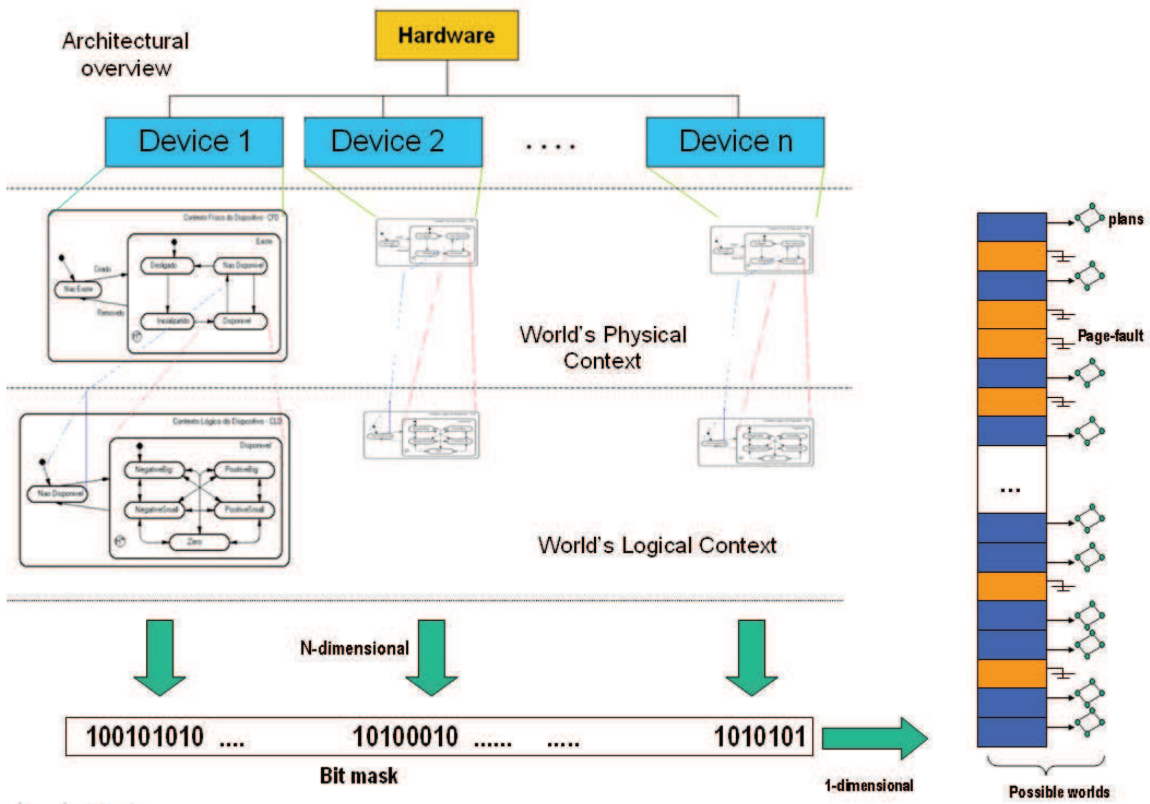


Fig. 4. A KBOS world model in an overall perspective

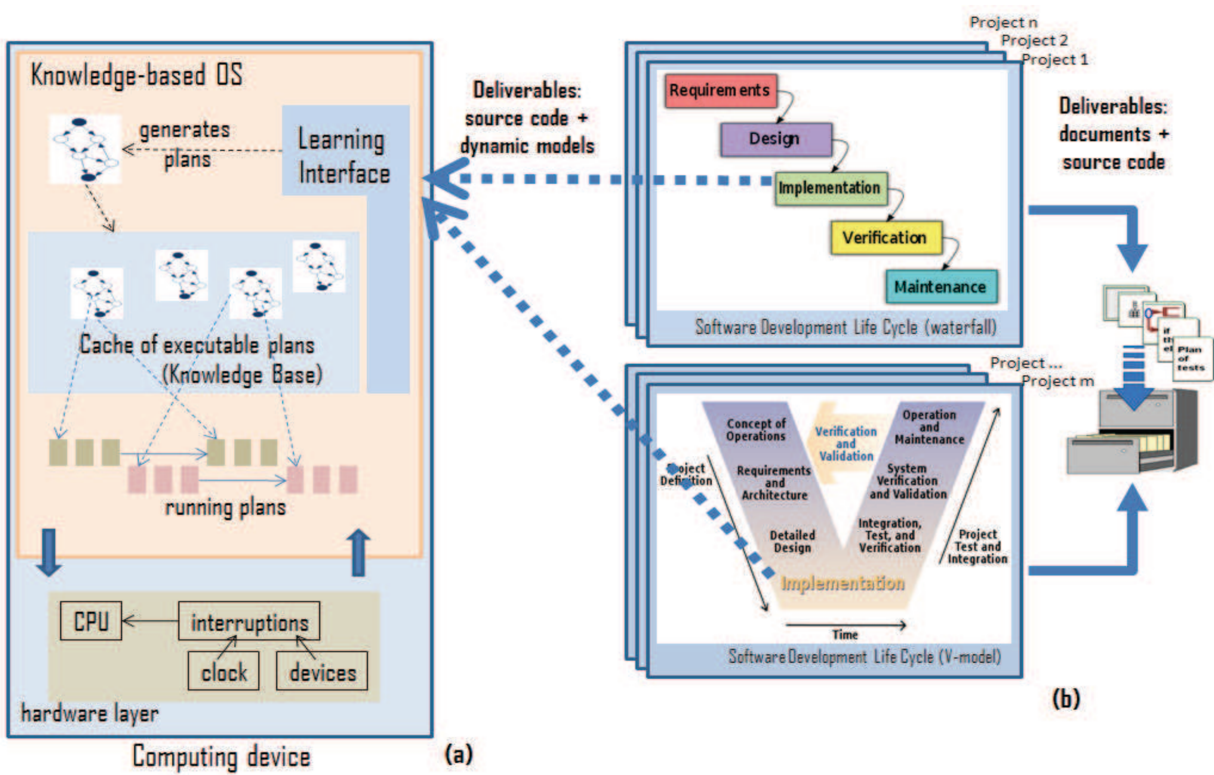


Fig. 5. Overview of the proposed software development process.

We should point some important characteristics:

- Knowledge = source code + dynamic models: traditionally, the last phase of compiling process is the binary code generation. An executable code doesn't carries additional information about the intentions of a particular program. In function of that, operating systems has to be prepared for both – well behavior and non-conforming programs;
- Learning interface: in a KBOS environment the assimilation interface grabs the input pack (source code + dynamic models) and analyses its own knowledge base in order to identify similar procedures. If it finds two different solutions for the same problem it could produce plans to be evaluated by itself in order to identify which one is the best to be adopted. The outcome is a new set of plans to be inserted in its knowledge base.
- Executable plans (Schaad, 1998) instead programs: in our point of view, at same time that the program represents the programmer's knowledge about some domain, it also carries no additional information when executed by some computing device. In a KBOS, we bring to the environment that knowledge and make it available to the OS.

Traditionally, a plan is regarded as an ordered collection of executable primitives, or macros, that are decomposable into primitives. We have chosen decision trees as a plan representation structure in KBOS in function of a number of distinct advantages over other representations for reactive plans (Schaad, 1998):

- simplicity: decision trees are easy to implement in any programming language and the associated run-time system can also be simple;
- efficiency: decision trees execute very efficiently.
- stepped execution model: decision trees are a natural fit with the stepped, ex-ante arbitrated execution model and with the design principle of improvisation underlying it
- transparency: decision trees are easy to understand and debug
- layering: layering is important for expressing temporal coherence, such as persistence and sequences, and for code reuse.

In this context, a plan is a data structure that maps a state machine and the program source code to a set of parallel functional decision trees (PFDT) using the notation described in (Schaad, 1998). A plan consists of a set of instructions expressed in the notation of PFDT - the *steppables* (figure 6a). Each plan is encapsulated by an envelope, which, among other things allows the recording of information about the context of the world (physical, behavioral and temporal) at any given time.

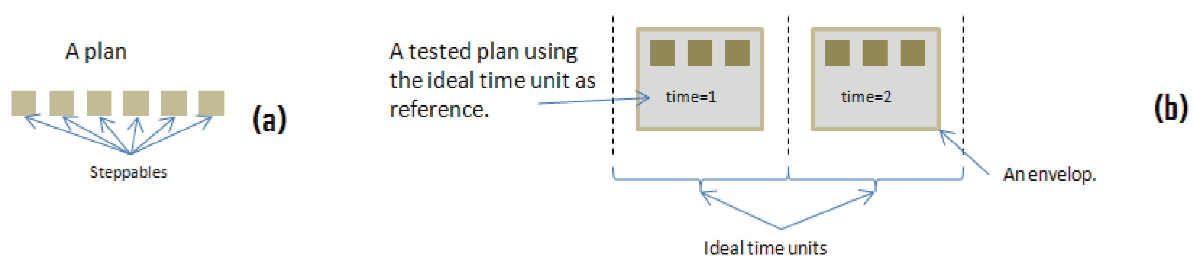


Fig. 6. An execution plan (a), and the plan tested in an ideal development environment (b)

The development of applications for KBOS introduces a requirement that the developer use an ideal development environment – an environment where it is possible to implement plans without interference from other applications/tasks (equivalent to running a program in a single-TASK operating system). This ideal environment has an abstract clock unit which

is used only for purposes of temporal ordering of the plans. The recording the execution time of a plan is a matter of increment the time counter (figure 6b).
If a plan is completely executed during an abstract interval, we register the value 1 as the time that plan needs to complete its task. Otherwise, if a plan requires more than one time unit, for each subsequent block of commands, that fits inside a time unit, we will increment this value until we get the end of the plan – observing that this procedure happens only at testing/debugging time.
In the situation where some sub-plans (any path within a program) have not been validated by the testing phase, the time of this path is recorded as invalid (-1). This information will allow the KBOS to become aware that the path was not previously tested and makes the KBOS to switch to a stage of learning.
At this stage, the plan being executed is monitored to assess effects on other plans being executed. As time passes and newer executions of this plan does not cause side effects, the plan starts being promoted to a condition in which he is regarded as reliable. Thus, one of the ways the KBOS acquires knowledge about the effects of some plan's behavior is by reinforcement learning.
If some error condition is detected, the system can provide to programmer the set of envelops with information about the environmental conditions at the error time. This adds important information about the timing in which the error occurred. Naturally it is not our expectation that a KBOS will develop the ability to correct programming logic errors.

4.2 The time dimension
Different definitions of time granularity have been proposed in the literature. All these definitions use partitions of a fixed temporal domain to represent temporal structures (Clifford & Rao, 1987), (Puppis, 2006).
In the current paradigm of computing systems, the time is a variable that has to be explicitly read (*get-time()/get-date() functions*) in order to enable software entities to perceive the time flow. The structure that we propose for the temporal domain is suitable for the needs of a KBOS and it is represented by a quadtree-like bit structure which enables to represent since the smallest observable or interesting time unit as well as a very coarse granularity (fig. 7).

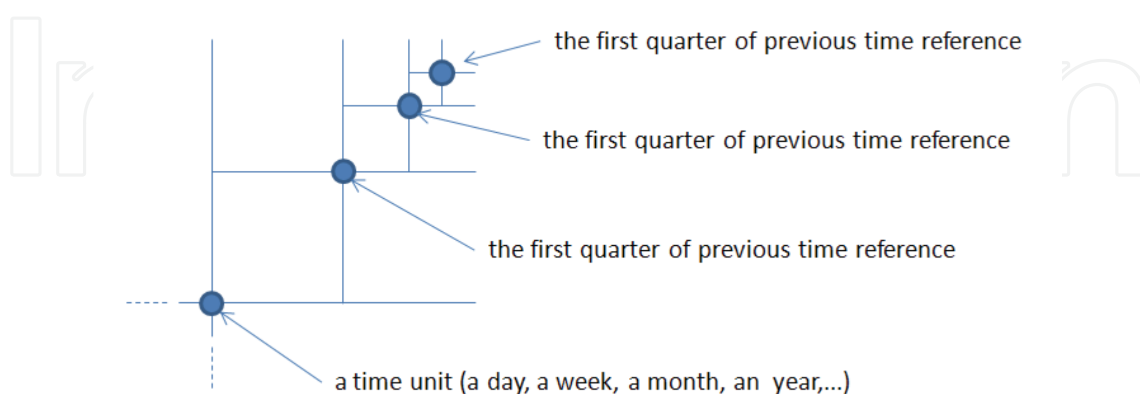


Fig. 7. The quadtree-like structure for representing time units (sub-units) in KBOS.

The term quadtree is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space.

Hierarchical data structures are useful because of their ability to focus on the interesting subsets of the data, resulting in an efficient representation and improved execution times, and it is thus particularly useful for performing set operations. Hierarchical data structures are attractive because of their conceptual clarity and ease of implementation (Samet, 1984).
This model allows to record and check the occurrence of an event on various time scales (figure 8) using the same universal structure, and with a very small computational cost. In this context, the manipulation of events is just a matter of bitwise operations (AND, OR, XOR, NOT) against the structure.

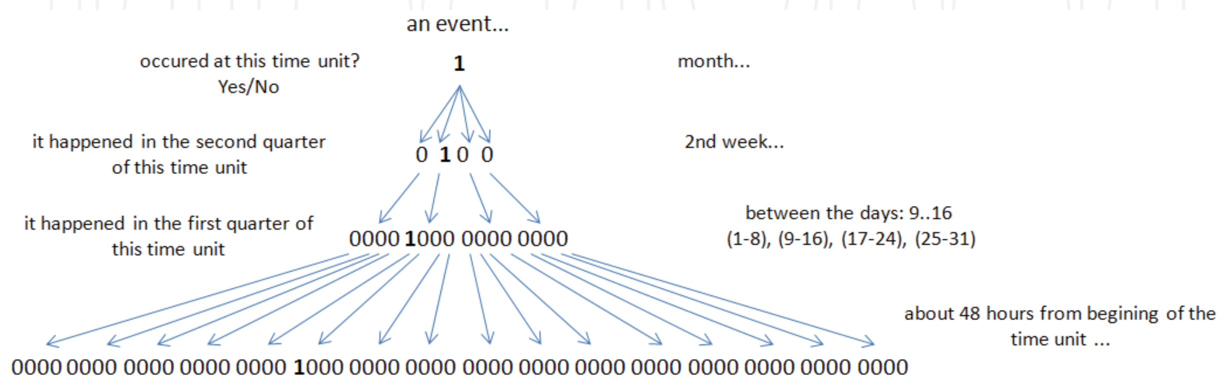


Fig. 8. Recording the occurrence of an event

There is a single global structure to represent the time for all instances of execution plans. Each execution plan can use sub-structures of the overall structure to represent time units in the application's domain. Thus, a reference to a specific unit of time is characterized as an index in this structure.
Once we have identified an universal structure to represent time units, the next step was to define the machine biological clock (MBC) and its computational representation.

4.3 Machine biological clock

As seen before, living organisms have some sort of internal biological clock that helps to define what is called: the circadian rhythm - the rhythm used to synchronize the internal actions within the body.

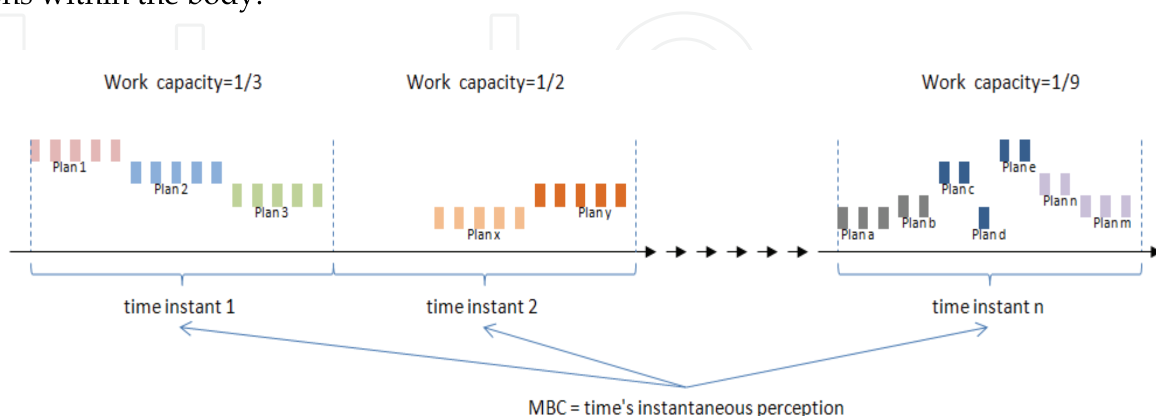


Fig. 9. Machine biological clock and work capacity

The MBC follows the same principle - a unit of time observable in the time's system structure (figure 9). However, we should to highlight that the events and actions taken

during this time interval are considered as *instantaneous* in such a way that a KBOS can only to consider this timescale for inference purposes. For analogy, in general, people do not perceive time units smaller than 1 second but we know that the internal functioning of the body works in smaller fractions of a second. In the same way, a KBOS is still capable of performing operations in fractions of seconds according to the characteristics of hardware but it will only perceive the time flow from the MBC units.

The MBC is a unit of time derived from the real time clock of the computing device. Thus, the size of the MBC (in units of fractions of seconds) is a matter of individual adjustment - each class of devices that share the same hardware characteristics should also share the same MBC. In the same way, different classes should have different MBC.

From the MBC concept it is possible to derive the concept of *work capacity* (WC) - a unit that measures the device's ability to perform tasks, which is measured in units of MBC (fig.8).

As more tasks needs to be executed by unit of MBC, less WC the device will present, and vice-versa.

This characteristic allows the system perceive that it is in an overcharged situation and this "feeling" will be propagated for all active plans instantly- i.e. in the time interval between two MBC time units. In this moment, all plans automatically will start to adapt themselves to that fluctuation condition. In the current paradigm, this situation could be evidenced, for example, because the queue of processes is long, or because the rate of context switching is high. However, if all processes are not context-aware, the system cannot adapt easily.

In a fairly high level of abstraction, this unit of *work capacity* enables a KBOS to make decisions when interacting with other devices in a community(Goumopoulos & Kameas, 2009) and to discover "*how good it is*" when comparing with the neighborhood devices - again, this characteristic could lead to some kind of measure of "*social behavior*" of the machine.

4.4 Time perception

Perceiving a time flow is a matter of to be situated. In order to achieve this goal, we had to conceive a complementary data structure to PFDT, refered before as envelop. We will depict a sample of how we made possible for plans to perceive the time flow without the need of explicitly asking it to the operating system.

The figure 10a shows a plan previously tested at the development environment by the programmer. This plan when comes to the user's machine and is assimilated will have the time units adjusted to the real MBC of the target device.

The figure 10b shows the plan being executed in a real situation sharing the cpu time with other plans but running according the time units previously defined.

The figure 10c shows a tipical situation where the availability of cpu is reduced by the fact the system needs to execute more plans. In this case, some part of the first envelop is sliced and scheduled to be executed in a time further. When the unfinished part of the first envelop starts running, its time recorded will be different from the actual MBC making with all further function calls be made with an indication of a delayed situation.

To a better understanding, the figure 11 shows an example of a java program that is time dependent as a sample of how we deal with time in the current paradigm.

In general, the program needs to call the *System.currentTimeMillis()* function in order to discover the current time and make some calculation to discover if it is delayed, on time or ahead of time (when comparing with previous execution of the same plan). Also, in general,

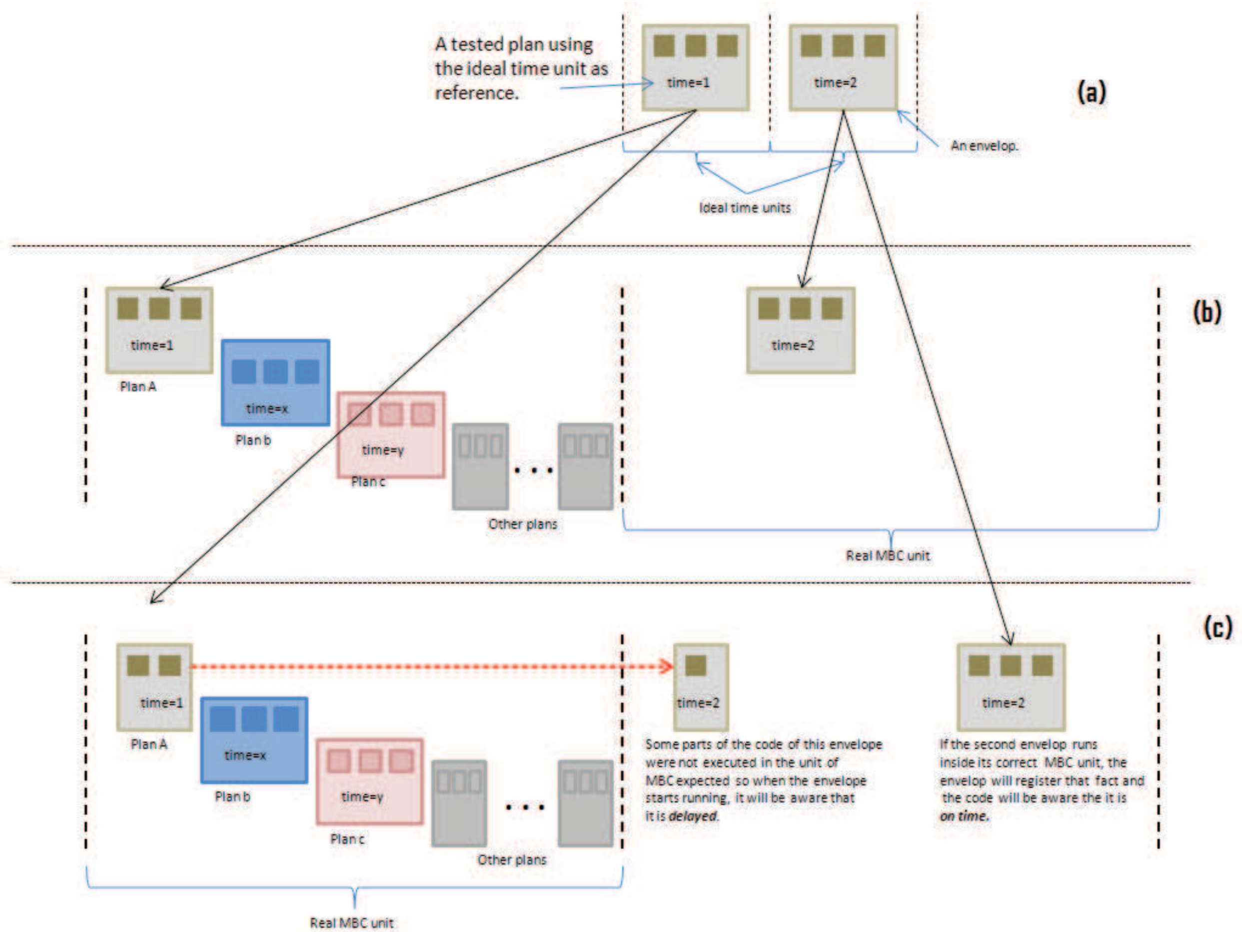


Fig. 10. A tested plan (a), the plan running in a real environment (b), and a plan perceiving it is delayed (c).

```
 6  public class AdaptationWithExplicitTimeManagement {
 7     private long deadline1, deadline2;
 8     private boolean hasSomethingToDo = true;
 9     public void doSomethingLate() {  }
10     public void doSomethingExactlyOnTime() {  }
11     public void doSomethingWithTimeLeft() {  }
12     public void processChanges1() {
13        while (hasSomethingToDo) {
14           if (System.currentTimeMillis() > deadline1)      doSomethingLate();
15           else if (System.currentTimeMillis() < deadline1) doSomethingWithTimeLeft();
16           else                                             doSomethingExactlyOnTime();
17        }//while
18     }
19     public void processChanges2() {
20        while (hasSomethingToDo) {
21           if (System.currentTimeMillis() > deadline2)      doSomethingLate();
22           else if (System.currentTimeMillis() < deadline2) doSomethingWithTimeLeft();
23           else                                             doSomethingExactlyOnTime();
24        }//while
25     }
```

Fig. 11. Explicit time example

only some portions of the code running in a system have to deal with such constraints, so we have a mix of code dealing with time and code that was not conceived to deal with time running together in the same environment. For example, the method *doSomethingExactlyOnTime()* could be started exactly on time, but become late because the multitasking environment could be scheduled other code to run changing the time when *doSomethingExactlyOnTime()* will effectively receive the CPU. If the code structure of that method is not build as the sample code (lines 14-16), the late execution could cause the entire system to present some strange behavior. And we are not talking about real-time applications, but desktop ones.

The figure 12 shows how a plan (in KBOS context) deal with time perception: each procedure/function has to explicitly declare sections where the time dimension has to be considered as a functional requisite.

In this example we can observe that there are three methods implementing the logic for *processChanges()* each one ending with one of the reserved words: *_Late*, *_OnTime* and *_AheadOfTime*.

During the execution of a plan, the system activates the appropriated section (*late*, *onTime* or *aheadOfTime*) according to the situation of the world model as previously described. If the developer does not know what to do in some situation, he can explicitly use an *IDoNotKnowWhatToDo* clause and the KBOS run-time will start trying to learn how to deal with that situation. This leads to some possibilities:

- the knowledge-base already have some other plan that already was tested before – this plan is activated;
- the knowledge-base does not have other plan – then the KBOS starts to follow the execution of the plan verifying what happens, for example, if the plan is delayed or cancelled.

In the example (figure 10), the method *processChanges1…()* was developed dealing with the three situations but in the situation where method *processChanges2…()* is dispatched late, the programmer explicitly declared that *iDontKnowWhatToDoInThisSituation()* (on line 20).

To support this model of function dispatch, the KBOS adopts a mapping function that makes possible to convert a n-dimensional world status to an one-dimensional status word The way we have implemented this functionality is changing the standard procedure call protocol:

call [memory address].

Instead, we have adopted the protocol:

call [memory address [World Status Word, CurrentMBC]].

When the destination address points to an *iDontKnowWhatToDoInThisSituation*, the system switches to a learning state. Under this condition, the KBOS can decide to let the procedure/function called run in one of the other possibilities programmed, or to cancel the call once the “programmer” doesn’t know how his program could behave under that condition.

If the system let the procedure/function call to continue, the system will learn what happens and will register the behavior of that part of the code under inadequate conditions in the PLAN’S state machine. So that the system will develop the capacity of observing the way a module works in order to decide, in the future, if it will allow or not that code to run again. Notice that this cannot grant that the code will do it right in the future.

www.intechopen.com

```
6   public class AdaptationWithImplicitTimeManagement {
7     private boolean hasSomethingToDo = true;
8     public void doSomethingLate() {  }
9     public void doSomethingExactlyOnTime() {  }
10    public void doSomethingWithTimeLeft() {  }
11
12    public void processChanges1_Late() {
13      while (hasSomethingToDo) { doSomethingLate(); }  }
14    public void processChanges1_OnTime()  {
15      while (hasSomethingToDo) { doSomethingExactlyOnTime(); }}
16    public void processChanges1_AheadOfTime()   {
17      while (hasSomethingToDo) { doSomethingWithTimeLeft(); }}
18
19    public void processChanges2_Late() {
20      iDontKnowWhatToDoInThisSitutation(); }
21    public void processChanges2_OnTime() {
22      while (hasSomethingToDo) { doSomethingExactlyOnTime(); }}
23    public void processChanges2_AheadOfTime() {
24      while (hasSomethingToDo) { doSomethingWithTimeLeft();} }
```

Fig. 12. Implicit time example.

The possibility to deal with time perception information is made explicit to the programmer in development time so, it allows him to make decisions for building/conceiving self-adaptive plans. On the other hand this introduces an additional level of difficulty because the programmers are not traditionally accustomed to thinking of time dimension in the conception of their programs.

The time dimension also makes it possible to introduce the vague notion of space concept if two different plans perceive that both are delayed it is equivalent to say to each one that there is someone else sharing resources within the same MBC unit. This could leads, for example, that the logical path of some plan could be changed to another path that implements the same functionality but demands less resources. The self-adaptive and self-reconfigurable characteristics of the system are based on this facility.

5. Related work

In general, the operating system designers are concerned primarily with problems of a purely quantitative nature (e.g. performance) (Hansen, 2000) (Peng, Li, & Mili, 2007) while qualitative aspects should receive more attention. Before continuing, it is important to look at the efforts already made towards changing the situation presented. We emphasize that we are excluding from this analysis those works aimed at improving the current model like (Hunt & Larus, 2007) (Lee, et al., 2010) mainly because we are interested in going deeper in the area of knowledge-based systems at operating system level.

By reviewing the literature, it is possible to find some references to a knowledge-based operating system. ((Sansonnet, Castan, Percebois, Botella, & Perez, 1982),(Vilensky, Arens, & Chin, 1984),(Blair, Mariani, Nicol, & Shepherd, 1987),(Chikayama, Sato, & Miyazaki, 1988),(Moon, 1985),(Larner, 1990),(Ali & Karlsson, 1990), (Xie, Du, Chen, Zheng, & Sun, 1995),(Patki, Raghunathan, & Khurshid, 1997), (Jankowski & Skowron, 2007)). Other approaches involve the application of artificial intelligence techniques through kernel

implants[6] to achieve better interfaces in traditional operating systems ((Pasquale, 1987), (Chu, Delp, Jamieson, Siegel, & Whinston, 1989), (Zomaya, Clements, & Olariu, 1998);(Kandel, Zhng, & Henne, 1998); (Holyer & Pehlivan, 2000),(Lim & Cho, 2007)).
However, all failed to achieve its objectives because the conceptual basis for the meaning of "knowledge" or "intelligence" was not properly established. In general, due to project's time constraints, prototypes are constructed using existing and proven technologies wherever possible, rather than implementing core technologies from scratch.
In (Stulp & Beetz, 2006) was proposed a novel computational model for the acquisition and application of action awareness, showing that it can be obtained by learning predictive action models from observed experience and also demonstrating how action awareness can be used to optimize, transform and coordinate underspecified plans with highly parametrizable actions in the context of robotic soccer. The system works in two moments:
i. idle time, when the agent learns prediction models from the actions in the action library; and
ii. operation time, when action chains are generated.
In (Tannenbaum, 2009) we found that self-awareness means learned behaviors that emerge in organisms whose brains have a sufficiently integrated, complex ability for associative learning and memory. Continual sensory input of information related to the organism causes its brain to learn its (the organism's) physical characteristics, and produce neural pathways, which come to be reinforced, so that the organism starts recognizing, several features associated to each reinforced pathway. The self-image characteristic provides a mechanistic basis for the rise of the concept of emergency of behavior that, on its turn, is connected to the concepts of self-awareness and self-recognition. On the basis of all that process there is the notion of time perception.

6. Conclusion

We have given an overview of an endogenous self-adaptive and self-reconfigurable approach to operating system design that we call: knowledge-based operating system.
In order to get there, we presented some evidences that lead us to go back in the origins of the modern computing and figure out what could be the reasons why we still are dealing with problems identified a long time ago.
In our point of view, the concepts of program, multitasking and operator are strong candidates to be considered.
A *program* is a rigid expression of the programmer's knowledge acquired during the software development life cycle that is transformed into a string of bits and expected to be managed by the OS. As the hardware was expensive, the OS designers found in the *multitasking* a way to better share the computational resources between different users. The *operator* was needed in order to make the installation ready for all users demanding computational resources.
We do not eliminate the concepts of program multitasking and operator, but instead, repositioned these concepts in the perspective of:

- a program shall be replaced by a plan of execution, whose code is generated internally by the system and externally to the user's environment by the traditional process of generating executable code; this changes the perspective of software setup towards a software learning;

[6] See (Seltzer, Small, & Smith, 1995) for a kernel implant explanation.

- The concept of multitasking becomes a tool to support the concept of MBC in that it now plans are explicitly able to: (a) perceive when they were sliced, and (b) perceives the fluctuation of resources availability of the computing device.

This characteristic allows the conception and development of really context-aware applications.
Insofar as the characteristics of adaptability become part of the system, the characteristics of the user-machine relationship become enriched.
We demonstrate that the concept of knowledge in this phase of the project is a matter of self-knowledge, or the computing device knowledge about its ability to perform tasks and to self-adapt to the fluctuations of resource availability in the environment.
To the extent that context-aware applications begin to take into account these characteristics, a new concept of intelligence and perception of intelligent behavior becomes evident.
In the context of this work, the role of the programmer now has a double function:

- on the one hand, it continues to map the knowledge of some application's domain for an encoding tool;
- on the other hand, it assumes the role of teaching the system how to perform the application's role.

This relationship expands the possibilities of what we call today the reuse of code for the reuse of knowledge.
Based on what was presented we could start thinking in terms of the machine's identity concept, which is resultant from the embodiment, situatedness, adaptiveness and autonomic characteristics of a KBOS. This leads to the emergence concept - a property of a total system which cannot be derived from the simple summation of properties of its constituent subsystems.
In this sense, the set of characteristics enables the system to perceive, in an individualized manner, a set of events occurring in some instant of time. Thus, the intelligent behavior emerges from the previous characteristics plus the relationship between the system and the surrounding environment (Müller-Schloer, 2004).
We believe that the major contribution of this work has been to present a new way of designing systems that can evolve in a natural way for the machines (Costa, 1993) opening an avenue for research in conceiving really embodied software artifacts on the context of ubiquitous computing environment.

7. References

Abowd, G. D., & Mynatt, E. D. (2000, March). Charting Past, Present, and Future Research in Ubiquitous Computing. Transactions on Computer-Human Interaction, 7, pp. 29-58.

Ackermann, T. (2010). Quantifying Risks in Service Networks:Using Probability Distributions for the Evaluation of Optimal Security Levels. AMCIS 2010 Proceedings.

Ali, K. A., & Karlsson, R. (1990). The Muse Or-parallel Prolog model and its performance. In S. Debray, & M. Hermenegildo (Ed.), Proceedings of the 1990 North American Conference on Logic Programming (Austin, Texas, USA) (pp. 757-776). Cambridge, MA: MIT Press.

Anderson, J. P. (1972). Computer Security Technology Planning Study Vol. 1. HQ Eletctronic Systems Division (AFSC), Deputy for Command and Management Systems, Bedford,Massachusetts.

Arnold, W. R., & Bowie, J. S. (1985). Artificial intelligence: a personal, commonsense journey. Upper Saddle River, NJ, USA: Prentice-Hall.

Balasubramaniam, D., Morrison, R., Kirby, G., Mickan, K., Warboys, B., Robertson, I., et al. (2005). A software architecture approach for structuring autonomic systems. ACM SIGSOFT Software Engineering Notes, 30 (4), 1-7.

Barham, P., Isaacs, R., Mortier, R., & Harris, T. (2006). Learning Communitacion Patterns in Sigularity. First Workshop on Tackling Computing Systems Problems with Machine Learning Techniques (SysML) - Co-located with SIGMETRICS 2006. Saint-Malo, France.

Bellman, K. L., Landauer, C., & Nelson, P. R. (2008). Systems Engineering for Organic Computing. In R. P. Würtz (Ed.), Understanding Complex Systems (p. 355). Bochum, Germany: Springer-Werlag.

Biba, E. (2010, March 01). Physicist Sean Carroll on "What is time"? Retrieved July 15, 2010, from Wired Science: http://www.wired.co.uk/news/archive/2010-03/01/physicist-sean-carroll-on-what-is-time

Blair, G. S., Mariani, J. A., Nicol, J. R., & Shepherd, D. (1987). A Knowledge-base Operating System. The Computer Journal, 30 (3), 193-200.

Brachman, R. J. (2002, Nov/Dec). Systems that know what they're doing. IEEE Intelligent Systems, 67-71.

Buschmann, F. (2010, September-October). On architecture styles and paradigms. IEEE Software, 92-94.

Carroll, S. M. (2008). What if Time Really Exists? arXiv:0811.3772v1 .

Chikayama, T., Sato, H., & Miyazaki, T. (1988). Overview of the parallel inference machine operating system (PIMOS). Proceedings of the International Conference of Fifth Generation Computer Systems., pp. 230-251.

Chu, C. H., Delp, E. J., Jamieson, L. H., Siegel, H. J., & Whinston, A. B. (1989, June). A model for an intelligent operating system for executing image understanding tasks on a reconfigurable parallel architecture. Journal of Parallel and Distributed Computing, pp. 598-622.

Church, R. M. (2006). Behavioristic, cognitive, biological, and quantitative explanations of timing. In E. A. Wasserman, & T. R. Zentall (Eds.), Comparative cognition: Experimental explorations of animal intelligence. (pp. 24-269). New York, NY, EUA: Oxford University Press.

Costa, A. C. (1993). Inteligência de máquina: esboço de uma abordagem construtivista. Federal University of Rio Grande do Sul, Institute of Informatics, Porto Alegre, Brazil.

da Costa, C. A., Yamin, A. C., & Geyer, C. F. (2008). Toward a general software infrastructure for ubiquitous computing. IEEE Pervasive Computing, 7 (1), 64-73.

Davidsson, P. (1994). Autonomous Agents and the Concept of Concepts (Thesis). Lund University.

Davis, R., Shrobe, H., & Szolovits, P. (1993). What Is a Knowledge Representation? AI Magazine, 14 (1), 17-33.

Duch, W. (2007). What is computational intelligence and where is it going? Challenges for Computational Intelligence, 63, 1-13.

Fleisch, B. D. (1983). Operating systems: a perspective on future trends. SIGOPS Operating Systems Review, 17 (2), pp. 14-17.

Franklin, S., & Graesser, A. (1996). Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents. In J. P. Muller, M. Wooldridge, & N. R. Jennings (Eds.), Lecture Notes in Computer Science (Vol. 1193, pp. 21-35). London, UK: Springer-Verlag.

Frost, R. A. (1986). Introduction to Knowledge Base Systems. Collins.

Grimm, R., Davis, J., Lemar, E., Macbeth, A., Swanson, S., Tom, S. G., et al. (2001). System-level Programming Abstractions for Ubiquitous Computing. Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII).

GSLC. (2010, May 28). The time of our lives. Retrieved September 29, 2010, from Learn.Genetics - Genetic Science Learning Center: http://learn.genetics.utah.edu/content/begin/dna/clockgenes/

Haigh, T. (2002, January-March). Software in the 1960s as concept, service, and product. IEEE Annals of the History of Computing, 24 (1), pp. 5-13.

Hansen, P. B. (1973). Operating systems principles. Upper Saddle River, NJ, USA: Prentice-Hall,Inc.

Hansen, P. B. (1977). The architecture of concurrent programs. Upper Saddle River, NJ, USA: Prentice-Hall,Inc.

Hansen, P. B. (2000). The evolution of operating systems. In Classic operating systems: from batch processing to distributed systems (pp. 1-36). New York, NY, USA: Springer-Verlag New York, Inc.

Hayes-Roth, R. (2006). Puppetry vs. creationism: why AI must cross the chasm. IEEE Intelligent Systems, 21 (5), 7-9.

Holyer, I., & Pehlivan, H. (2000). A Recovery Mechanism for Shells. The Computer Journal, 43 (3), 168-176.

Hunt, G. C., & Larus, J. R. (2007). Singularity: rethinking the software stack. SIGOPS Operating Systems Review, 41 (2), 37-49.

Hunt, G., Larus, J., Abadi, M., Aiken, M., Barham, P., Fähndrich, M., et al. (2005). An overview of the Singularity project. Redmond, WA: Microsoft Research.

Iyoengar, K., Sachdev, V., & Raja, M. K. (2007). A security comparison of open-source and closed-source operating systems. Proceedings of South West Decision Sciences Thirty-eighth Anual Conference. San Diego, CA, USA.

Jankowski, A., & Skowron, A. (2007). A Wistech Paradigm for Intelligent Systems. (J. Peters, A. Skowron, I. Düntsch, J. Grzymala-Busse, E. Orlowska, & L. Polkowski, Eds.) Lecture Notes in Computer Science:Transactions on Rough Sets VI, 4374, pp. 94-132.

Josephson, J. R., & Josephson, S. G. (1994). Abductive Inference:Computation, Philosophy,Technology. New York: Cambridge University Press.

Kandel, A., Zhng, Y.-Q., & Henne, M. (1998). On use of fuzzy logic technology in operating systems. Fuzzy Sets and Systems, 99 (3), 241-251.

Klausman, E. F. (1961). Training the computer operator. ACM '61:Proceedings of the 1961 16th ACM national meeting (pp. 131.401-131.404). New York, NY, USA: ACM.

Kramer, J., & Magee, J. (2007). Self-Managed Systems: an Architectural Challenge. 2007 Future of Software Engineering (May 23 - 25, 2007). International Conference on Software Engineering (pp. 259-268). Washington, DC, EUA: IEEE Computer Society.

Kruchten, P., Capilla, R., & Dueñas, J. C. (2009). The decision view's role in software architecture practice. IEEE Software, 26 (2), 36-42.

Krutchen, P. (2004). An Ontology of Architectural Design Decisions in Software Intensive Systems. 2nd Groningen Workshop Software Variability, (pp. 54-61).

Kupsch, J. A., & Miller, B. P. (2009). Manual vs. automated vulnerability assessment: a case study. First International Workshop on Managing Insider Security Threats (MIST 2009). West Lafayette, IN.

Lagerström, R., von Würtemberg, L. M., Holm, H., & Luczak, O. (2010). Identifying factors affecting software development cost. Proc. of the Fourth International Workshop of Software Quality and Maintainability (SQM).

Larner, D. L. (1990). A distributed, operating system based, blackboard architecture for real-time control. IEA/AIE '90: Proceedings of the 3rd international conference on Industrial and engineering applications of artificial intelligence and expert systems (pp. 99-108). Charleston, South Carolina, US: ACM.

Lee, S.-M., Suh, S.-B., Jeong, B., Mo, S., Jung, B. M., Yoo, J.-H., et al. (2010). Fine-grained I/O access control of the mobile devices based on the Xen architecture. of the 15th Annual international Conference on Mobile Computing and Networking (Beijing, China, September 20 - 25, 2009) (pp. 273-284). Beijing, China: ACM, NY, USA.

Legg, S., & Hutter, M. (2007, December). Universal intelligence: a definition of machine intelligence. Minds and Machines, pp. 391-444.

Lenat, D. B., & Feigenbaum, E. A. (1988). On the thresholds of knowledge. Proceedings of the International Workshop on Artificial Intelligence for Industrial Applications (IEEE AI'88), (pp. 291-300). Hitachi City, Japan.

Lim, S., & Cho, S.-B. (2007). Intelligent OS process scheduling using fuzzy inference with user models. IEA/AIE'07: Proceedings of the 20th international conference on Industrial, engineering, and other applications of applied intelligent systems (pp. 725-234). Kyoto, Japan: Springer-Verlag.

Linde, R. (1975). Operating Systems Penetration. AFIPS Conference Proceedings, 44.

Malsburg, C. v. (2008). The Organic Future of Information Technology. In R. P. Würtz (Ed.), Understanding Complex Systems (p. 355). Bochum: Springer-Verlag.

Mattos, M. M. (2003). Fundamentos conceituais para a construção de sistemas operacionais baseados em conhecimento (thesis). thesis, UFSC - Universidade Federal de Santa Catarina, PPGEP - Programa de Pós-Graduação em Engenharia de Produção, Florianópolis, Brazil.

Meadow, C. T., & Yuan, W. (1997). Measuring the impact of information: defining the concepts. Information Processing & Management, 33 (6), 697-714.

Milner, R. (2006, March). Ubiquitous computing: shall we understand it? The Computer Journal, 383-389.

Moon, D. A. (1985, June). Architecture of the Symbolics 3600. SIGARCH Computer Architecture News., 13 (3), pp. 76-83.

Müller-Schloer, C. (2004). Organic omputing: on the feasibility of controlled emergence. 2nd IEEE/ACM/IFIP International Conference on Hardware e Software Codesign and System Synthesis. Stockholm,Sweden: ACM Press.

Murch, R. (2004). Autonomic Computing. IBM Press.

Naur, P., & Randell, B. (1969). Software Engineering: Report of a Conference Sponsored by the NATO Science Committee. In P. Naur, & B. Randell (Ed.). (p. 136). Garmisch, Germany: Scinetific Affairs Division, NATO.

Osterweil, L. J. (2007). A future for software engineering? FOSE '07: 2007 Future of Software Engineering (May 23-25,2007) International Conference on Software Engineering (pp. 1-11). Washington, DC, USA: IEEE Computer Society.

Overton, W. F. (1994). The arrow of time and the cycle of time: concepts of change, cognition, and embodiment. (L. A. Pervin, Ed.) Psychological Inquiry: An International Journal of Peer Commentary and Review., 5 (3), 215-237.

Pasquale, J. C. (1987). Using expert systems to manage distributed computer systems. Berkeley, CA, USA: University of California at Berkeley.

Patki, A. B., Raghunathan, G. V., & Khurshid, A. (1997). FUZOS - Fuzzy Operating System Support for Information Technology. Proceedings of Second On-line World Conference on Soft Computing in Engineering,Design and Manufacturing. (pp. 23-27). Bedfordshire,UK: Cranfield University.

Peng, Y., Li, F., & Mili, A. (2007). Modeling the evolution of operating systems: an empirical study. Journal of Systems Software, 80 (1), 1-15.

Perry, D. E., & Wolf, A. L. (1992, October). Foundations for the study of software architecture. (ACM, Ed.) ACM SIGSOFT Software Engineering Notes, 17 (4), pp. 40-52.

Pfleeger, S. L. (2010, July/August). Anatomy of an Intrusion. IEEE IT Professional, 12 (4), pp. 20-28.

Poole, D., Mackworth, A., & Goebel, R. (1997). Computational intelligence: a logical approach. Oxford, UK: Oxford University Press.

Poslad, S. (2009). Smart Devices and Services, in Ubiquitous Computing: Smart Devices, Environments and Interactions. Chichester,UK: John Wiley & Sons, Ltd.

Post, G., & Kagan, A. (2003). Computer security and operating system updates. Information and Software Technology, 45 (8), 461-467.

Ragunath, P. K., Velmourougan, S., Davachelvan, P., Kayalvizhi, S., & Ravimohan, R. (2010, January). Evolving a new model (SDLC Model-2010) for software development life cycle (SDLC). International Journal of Computer Science and Network Security, 10 (1), pp. 112-119.

Randell, B. (1979). Software engineering in 1968. ICSE '79: Proceedings of the 4th international conference on Software engineering (pp. 1-10). Munich, Germany: IEEE Press.

Samek, M. (2009). Practical UML Statechars in C/C++, Second Edition: Event-Driven Programming for Embedded Systems (2a ed.). Newton, MA, USA: Newnes.

Sansonnet, J. P., Castan, M., Percebois, C., Botella, D., & Perez, J. (1982, March). Direct execution of lisp on a list-directed architecture. SIGARCH Computer Architecture News, 10 (2), pp. 132-139.

Schaad, R. (1998). Representation and Execution of Situated Action Sequences (Thesis). Universitat Zürich.

Schmidt, C., Collette, F., Cajochen, C., & Peigneux, P. (2007). A time to think: circadian rhythms in human cognition. Cognitive Neuropsychology, 24 (7), 755-789.

Seltzer, M., Small, C., & Smith, K. (1995). The case for extensible operating systems. Harward Computer Center for Research in Computing Technology.

Shadbolt, N., O'hara, K., & Crow, L. (1999, October). The experimental evaluation of knowledge acquisition techniques and methods: history, problems and new directions. International Journal of Human-Computer Studies, 51 (4), pp. 729-755.

Shibayama, S., Sakai, H., & Takewaki, T. (1988). Overview of knowledge base mechanism. Proceedings of the International Conference on Fifth Generation Computer Systems, pp. 197-207.

Skjellum, A., Dimitrov, R., Angaluri, S. V., Coulouris, G., Uthayopas, P., Scott, S. L., et al. (2001, May). Systems Administration. International Journal of High Performance Computing Applications, 15 (2), pp. 143-161.

Stenger, V. J. (2001). Time's arrows point both ways: the view from nowhen. Skeptic, 8 (4), 92-95.

Stulp, F., & Beetz, M. (2006). Action awareness – enabling agents to optimize, transform, and coordinate plans. Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS) (pp. 8-12). Hakodate, Hokkaido, Japan: ACM.

Tanenbaum, A. S. (2008). Modern Operating Systems. Prentice Hall.

Tannenbaum, E. (2009, June). Speculations on the emergence of self-awareness in big-brained organisms. Consciousness and Cognition, 18 (2), pp. 414-427.

Vilensky, R., Arens, Y., & Chin, D. (1984, June). Talking to UNIX in English: an overview of UC. Communications of ACM, 27 (6), pp. 574-593.

Weiser, M. (1991, September). The Computer for the Twenty-First Century. Scientific American, 94-10.

Wilensky, R., Chin, D. N., Luria, M., Martin, J., Mayfield, J., & Wu, D. (2000, April). The Berkeley UNIX Consultant Project. Artificial Intelligence Review, 14 (1-2), pp. 43-88.

Würtz, R. P. (2008). Introduction: Organic Computing. In R. P. Würtz (Ed.), Understanding Complex Systems (p. 355). Bochum, Germany: Springer-Verlag.

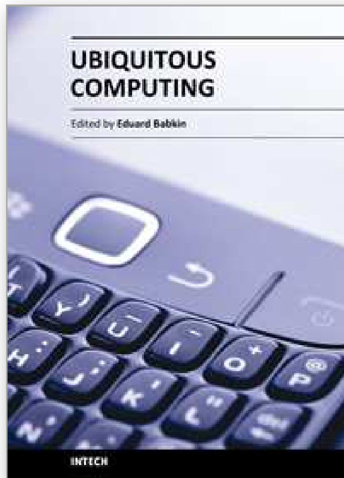
Xie, L., & Yi, J. (1998). A model for intelligent resource management in a large distributed system. Science in China Series E: Technological Sciences, 41 (1), pp. 13-21.

Xie, L., Du, X., Chen, J., Zheng, Y., & Sun, Z. (1995). An introduction to intelligent operating system KZ2. SIGOPS Operating Systems Review, 29 (1), pp. 29-46.

Yokote, Y. (1992). The Apertos reflective Operating System: the concept and its implementation. Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications -OOPSLA '92. 27, pp. 414-434. New York, NY,EUA: ACM.

Yuhua, Z., Honglei, T., & Li, X. (1993, May). And/Or parallel execution of logic programs: exploiting dependent And-parallelism. SIGPLAN Notices, pp. 19-28.

Zomaya, A. Y., Clements, M., & Olariu, S. (1998, March). A framework for reinforcement based scheduling in parallel processor systems. IEEE Transactions on Parallel and Distributed Systems., 249-260.



Ubiquitous Computing
Edited by Prof. Eduard Babkin

ISBN 978-953-307-409-2
Hard cover, 248 pages
Publisher InTech
Published online 10, February, 2011
Published in print edition February, 2011

The aim of this book is to give a treatment of the actively developed domain of Ubiquitous computing. Originally proposed by Mark D. Weiser, the concept of Ubiquitous computing enables a real-time global sensing, context-aware informational retrieval, multi-modal interaction with the user and enhanced visualization capabilities. In effect, Ubiquitous computing environments give extremely new and futuristic abilities to look at and interact with our habitat at any time and from anywhere. In that domain, researchers are confronted with many foundational, technological and engineering issues which were not known before. Detailed cross-disciplinary coverage of these issues is really needed today for further progress and widening of application range. This book collects twelve original works of researchers from eleven countries, which are clustered into four sections: Foundations, Security and Privacy, Integration and Middleware, Practical Applications.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Mauro Marcelo Mattos (2011). Machine Biological Clock: Exploring the Time Dimension in an Organic-Based Operating System, Ubiquitous Computing, Prof. Eduard Babkin (Ed.), ISBN: 978-953-307-409-2, InTech, Available from: http://www.intechopen.com/books/ubiquitous-computing/machine-biological-clock-exploring-the-time-dimension-in-an-organic-based-operating-system

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2011 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike-3.0 License, which permits use, distribution and reproduction for non-commercial purposes, provided the original is properly cited and derivative works building on this content are distributed under the same license.