PERGAMON

Annual Reviews in Control 25 (2001) 133–146

Annual Reviews in Control

www.elsevier.com/locate/arcontrol

Real-Time Software Architectures and Design Patterns: Fundamental Concepts and Their Consequences

Janusz Zalewski
School of Electrical Engineering & Computer Science
University of Central Florida
Orlando, FL 32816-2450, USA
jza@ece.engr.ucf.edu

Abstract. This paper discusses the principles of software architectures for real-time systems. The fundamental idea of a real-time architecture is based on the concept of feedback used in control engineering. A generic architecture is derived for four major categories of real-time systems. Then a fundamental design pattern is presented, valid for all relevant architectures. This is followed by a discussion of variations in the basic architecture for distributed systems and safety related systems. Finally, a case study and tool support for architectural design and implementation are discussed.
Keywords. Real-time systems, real-time computing, software architecture, design patterns, safety related systems, software tools, history of engineering.

1 Introduction

In recent years, a new area of research and engineering has emerged called software architecture [7, 11, 15, 34, 40, 47]. It is not clear, however, what are the principles of software architectures, what are good and bad examples or practices, and how to build high quality software according to certain principles and following good practices. The state of the art is such that almost every block diagram with interconnected boxes and a couple of arrows pointing from one box to another is claimed to be a description of a software architecture.

By any measure, this is far from being acceptable. Just like not every combination of construction materials can lead to a good house structure and not every combination of structural elements can lead to a good house architecture, there are only certain selective arrangements of software elements that can constitute a good software architecture. To find out what are these arrangements of elements and structures, which are crucial to software architectures, we have to look at the basic principles. We have to determine what constitutes an architecture of software. In particular, in this paper, we look at the issue of what constitutes an architecture of a real-time system software.

Architectures specific to real-time and related systems have been studied or proposed in a number of papers, in the last one and a half decade, for example [4, 6, 10, 18, 26, 36, 39]. Some of these papers propose interesting ideas to approach the real-time software architecture problem from the perspective of control engineering. This approach seems to be very promising and is pursued by us, but to be fully fruitful and understood it has to be tied to some more fundamental concepts.

There are certain fundamental laws of nature, which have to be taken into account by engineers. For example, for a house architecture, these laws of nature are dictated by the force of gravity. To start building a house from the roof would not be a good idea, even though such houses standing on a roof do exist, one in the author's hometown [50].

In the field of real-time systems, such a fundamental law exists as well, although its significance has not been sufficiently articulated. It is the feedback principle, one of the most fundamental laws, which both the nature and the society rely upon. Basically, every human being and every society can exist because of this fundamental feedback property, to mention only the most obvious examples: regulation of temperature of a human body and self-regulating principles of economy.

In this paper, the author claims that the fundamentals of real-time computing and real-time systems architectures are deeply rooted in the feedback principle, the basic law of control engineering, and date back to the B.C. era; at least that's how far they can be traced. The rest of the paper is structured as follows. Section 2 outlines these basic roots in the history of engineering. Sec-

1367-5788/01/$20 © 2001 Published by Elsevier Science Ltd.
PII: S1367-5788(01)00013-X

tion 3 derives the basic architecture of a real-time system, and Section 4 discusses the basic design patterns. Distributed real-time architectures and safety-related architectures are treated in Sections 5 and 6, respectively. An illustrative example is presented in Section 7 and principles of design and implementation tool support for real-time architectures are discussed in Sections 8 and 9.

2 Ancient Roots of Real-Time Computing

Historically, the most familiar example of an engineering application of the feedback principle is probably the device known as the centrifugal speed governor [8, 29]. In 1767, James Watt invented a flyball speed governor to control the speed of steam engines. In this device, two spinning flyballs fastened to a shaft rise as a rotational speed increases, causing a mechanically connected steam valve to close, thus reducing the steam flow to the engine and regulating the speed. The control feedback principle applied in this device is illustrated in Figure 1.

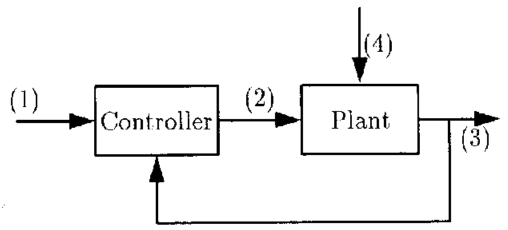

Fig. 1. Abstract view of the feedback principle (Watt's speed governor). (1) Desired value (speed set point); (2) Command signal (error value); (3) Controlled variable (actual speed); (4) Disturbances (load and steam pressure).

Intuitively, we would agree that the Watt's speed governor operates in real time: engine's action (that is, that of the 'plant') to increase or decrease the speed causes an immediate (that is, real-time) response of the controller to close or open the valve. Moreover, we can calculate how "immediate" are these actions, because the exact timing relationships do exist between the controller and the plant. This is described by the equations of the system dynamics. All this means that the engine speed governor can be used as a model of a simple real-time system. A straightforward analysis reveals that the major elements of a controller are:

- plant interface (sensor and actuator)
- user interface (set point)
- processing power (subtraction operation and other transformations within the controller).

Studying the history of engineering, in particular, control engineering [29], reveals that the oldest written record of an application of the feedback principle is that of a water clock, that is, a device to measure time. Presumably in the first half of the third century B.C., an inventive Greek craftsman, Ktesibios, invented a device which used a constant flow of water to measure the passage of time (Fig. 2). Water drops falling from the orifice of an upper tank accumulate in the lower tank and the increasing water level indicates how much time expired. However, the accuracy of the clock depends on the constant flow rate of water into the low-level container. This flow rate is being disturbed by variation in the upper-tank water level, that is, its water pressure. To solve the problem of variations in water flow rate, Ktesibios maintained a constant water level in the upper holding tank by applying a valve with a float. The role of the valve is to open and allow water flow in, when the water level in the upper tank goes down, and to close and cut the water stream off, when the water level in the upper tank reaches the uppermost level. The water level in the upper tank is determined (measured) by a float (a sensor) which operates as a part of the valve (an actuator).

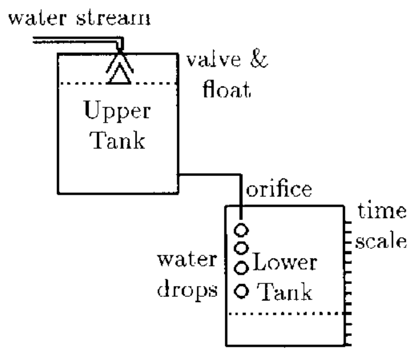

Fig. 2. Sketch of a water clock of Ktesibios (water level is marked by dotted lines).

This simple device invented in the third century B.C. fits very well into the model of feedback principle and system dynamics, and can be regarded as the first real-time device. In fact, it would be hard to find a better example for our purposes, because this device not only operates in real time but also serves the purpose of measuring (computing) time. Interestingly, the oldest preserved written source on Ktesibios' water clock is a book on architecture, by Vitruvius [46].

3 Generic Architecture

In modern real-time systems, which in principle, are control systems, in one way or another, a number of elements has been added due to the introduction of computers. The computational power changed the nature of the controller and the environment, but not the nature of the feedback principle. Building on the examples presented thus far, we can introduce computing power to the feedback principle diagram. This issue has been discussed by several authors studying relationships between computation and control [5, 9, 42].

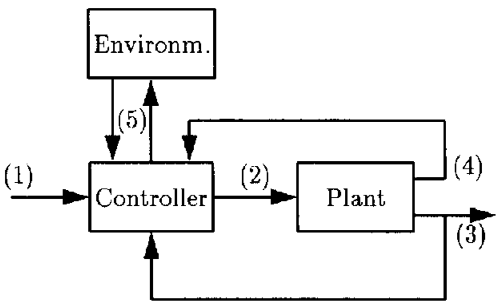

Fig. 3. Illustration of a control system. (1) Desired value; (2) Controller commands; (3) Controlled variables; (4) Other measured variables; (5) Environment interface.

An illustrative example is presented in Fig. 3. The roles of interface variables are described in more details below:

(1) Desired value; a reference for the Controller to make necessary adjustments of controlled variables.

(2) Controller commands; signals applied to the Plant (outputs from the Controller) in order to achieve its desired behavior.

(3) Controlled variables; signals received from the Plant (inputs to the Controller), whose values are being controlled.

(4) Other measured variables; auxiliary signals received from the plant (inputs to the Controller) which are not controlled but used in the determination of the best values of Controller commands.

(5) Environment interfaces – user interface, mass storage interface, communication link to computer network.

It is evident from the above that the modern controller became a digital processor or a real-time computer, and its interaction with the environment, in addition to that with sensors and actuators, includes interfaces with the following:

- plant operator
- computer network
- mass storage (database).

A unified diagram including explicitly all these elements is presented in Fig. 4.

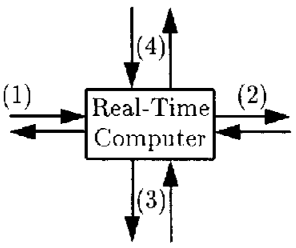

Fig. 4. Real-time computer system. (1) User interface; (2) Process interface; (3) Mass storage interface; (4) Communication link.

In practice, a number of real-time systems exist which do not represent a complete system in a sense of Fig. 3, but nevertheless fit very well into this concept. Respective examples are discussed in the following three sections.

3.1 Data Acquisition System

If the connection (2) is broken, in Fig. 3, one has a plain data acquisition system [19, 48]. There are virtually no controller commands sent to the plant, which results in a system shown in Fig. 5.

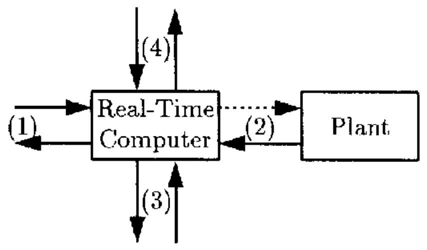

Fig. 5. Illustration of a data acquisition system (notation same as in Fig. 4).

The most prominent example of a data acquisition system is an air traffic control system [33]. Surprisingly, it is not, in fact, an automatic control system at all, because there is no direct connection between the real-time computer and the plant (airspace). All commands are executed by the pilot who is receiving respective messages from an air-traffic controller (Fig. 6).

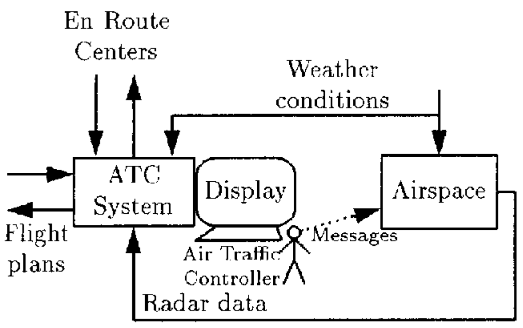

Fig. 6. Air traffic control system as a data acquisition system.

3.2 Programmed Controllers

If the feedback connection in Fig. 3, from the plant to the controller, is removed, with connection from the controller to the plant remaining intact, then we have a programmed control system (Fig. 7), for example, a traffic light control system, which in fact does not receive much feedback information. Simpler and more familiar examples of programmed control systems include a microwave oven controller and a washing machine controller. In principle, their control signals are precomputed in advance. All of these devices, however, continue to operate in real time.

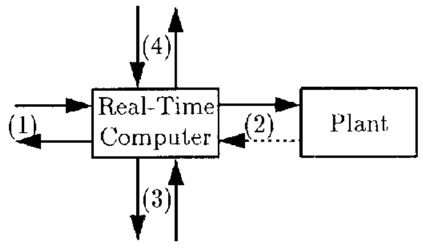

Fig. 7. Illustration of a programmed control (notation same as in Fig. 4).

For illustrative purposes, we can make a variation of a water clock which uses programmed control. This occurs, if we abondon the feedback principle but still require the water level to change with a constant rate to indicate proportionally the time elapsed. For this purpose we just need one tank. To find a solution to this problem, one has to calculate an appropriate shape of a single tank, in which the water level decreases constantly over time [17]. In the Cartesian coordinates (x, y), in ideal conditions, the shape of a tank can be described using Torricelli's law by the following differential equation relating the tank volume, V, and the shape of tank's walls in the vertical dimension, y:

$$\frac{dV}{dt} = -k\sqrt{y}$$

where a constant coefficient k depends on the exit area, discharge coefficient and gravitational acceleration.

On the other hand, the rate of change in water volume in the container, V, can be related to the shape of walls by the following equation:

$$\frac{dV}{dt} = A(y)\frac{dy}{dt}$$

where $A(y) = \pi x^2$ is the area of the water surface in the tank.

Comparing the right-hand sides and solving the resulting equation for $dy/dt = const$ we obtain the solution as the following formula:

$$y = f(x) = ax^4$$

for the shape of container walls, assuring us of the constant decrease in water level. The value of coefficient a can be obtained from other constants involved.

3.3 Reduced Architecture

Even if both connections between a plant and a real-time computer are broken, we can still claim that what remains is a reduced architecture for a valid real-time system (Fig. 8).

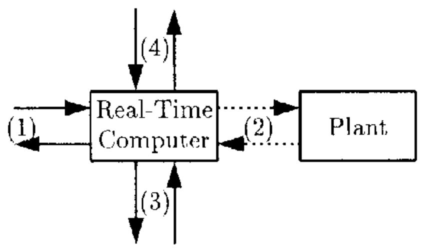

Fig. 8. Illustration of a reduced real-time architecture (notation same as in Fig. 4).

There are several practical examples of that kind of a real-time system. Putting emphasis on the distribution and communication, with relatively less interest in GUI and database access, brings us to a typical case of real-time simulation. With a slightly different emphasis, more on the database use and GUI, one has a real-time multimedia system.

4 Design Patterns

Once we understand the nature of an architecture of a real-time system, we can focus on developing its design and shaping its software architecture. It is at this point, when the concept of design patterns comes into play. What this means in practice is that our ability to apply engineering principles is significantly enhanced, because we can justify reliance on reusing proven existing solutions, commonly called design patterns.

The concept of design patterns becomes very clear if we take a look at other engineering disciplines. For example, there exists a clear pattern in designing radio receivers (Fig. 9). A radio receiver is always built out of certain components, such as an antenna, mixer and oscillator, detector, amplifier and speaker, connected in a predetermined way. It has been that way since radio receivers were conceived. What has changed and keeps changing is the technology and the implementations.

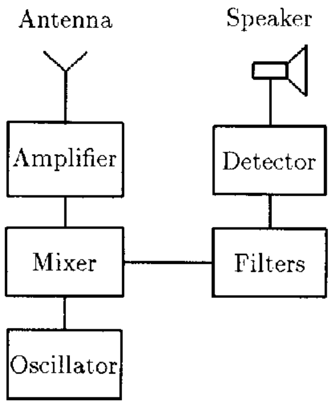

Fig. 9. Typical block diagram architecture of a radio receiver.

However, in real-time computing, somebody has to tell us what these major architectural components are and how are they related. Despite some well established concepts and principles of real-time computing [21, 41], thus far, there has been very little guidance on selecting real-time architectures, either in the engineering literature or in practice [13, 44]. However, when one takes a closer look at Figures 3 and 4, with explanations (1)-(5) in the previous section, there is little doubt that one can start and should start designing the controller from the context diagram similar to that in Fig. 10.

The role of a context diagram cannot be overestimated. Even though it is a relatively old notational vehicle, it's been well established in real-time software design as the basis for architectural development [23]. It is at the context diagram level, where the interfaces between the software and the external world need to be defined and developed. For this very reason, the concept of a context diagram is indespensible as a starting point in designing a software architecture.

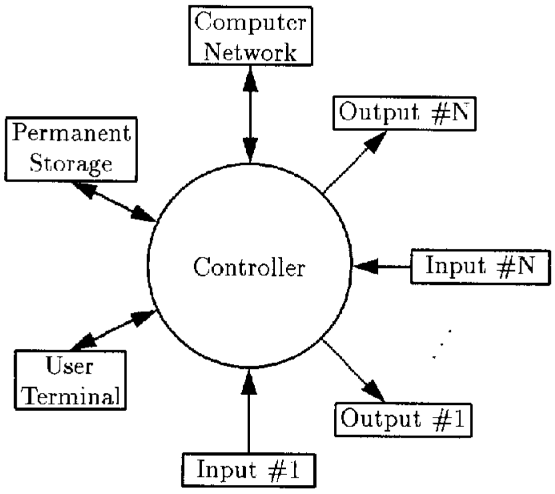

Fig. 10. The top level context diagram.

An example of the context diagram for the Air Traffic Control System (Fig. 6) is presented in Fig. 11.

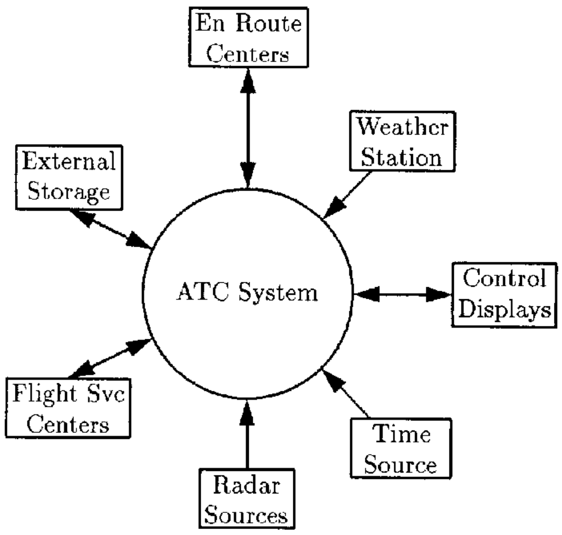

Fig. 11. The top level context diagram for the air traffic control system.

It is compatible with the generic context diagram shown above and includes:

- two functionally different kinds of mass storage interfaces (for flight plans and flight services)
- two functionally different kinds of network interfaces (for *en route* centers and weather services)
- two sources of data (radars and time source), as well as
- a user interface to controller displays.

From Figures 10 and 11, it becomes immediately clear that the software components must include parties responsible for the following interactions with all external elements:

- inputs from and outputs to the plant
- interaction with a user
- possible communication with other controllers/processors
- interaction with storage devices

enhanced by the processing (computational) capability. The time source can be internal or external depending on circumstances.

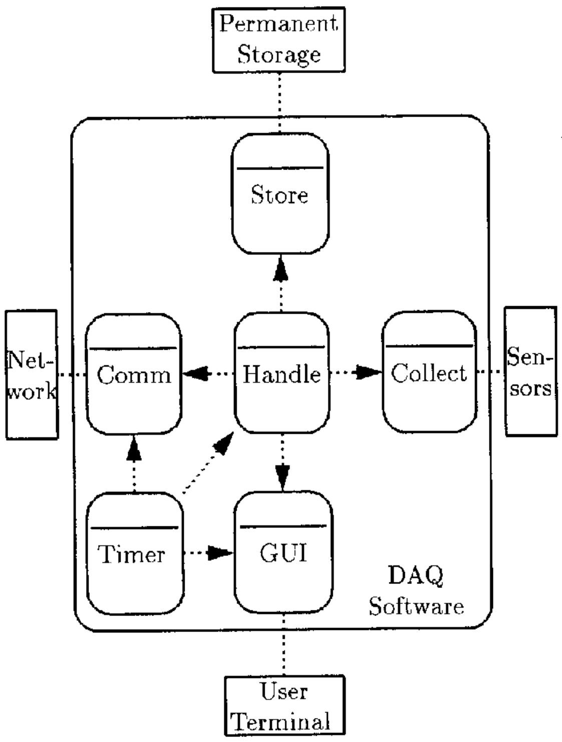

Fig. 12. Outline of a generic data acquisition system.

An example of such design, which can be considered a basic and generic design pattern for real-time systems software, is presented in Fig. 12, for a data acquisition application.

Respective software components need to comply with the principle of separation of concerns. They can be considered as sequential modules or individual concurrent tasks, and can run, respectively, on a single processor, on multiple processors, or even on a distributed system or network.

This basic design pattern can be expanded further into more comprehensive architectures, depending on the focus of particular applications. In the next sections, we consider the following additional issues, all derived from the basic concept of an architecture and design patterns:

- distributed real-time architectures [20]
- safety-related architectures [3, 26, 38]
- real-time design tools for architectural support.

5 Software Architecture of a Distributed System

Depending on what is the focus of a distributed real-time architecture, there may be a variety of its particular instances. One such example (Fig. 13) comes from the area of high-energy physics, where multiple data collection and control facilities are spread over a large area surrounding an elementary particle accelerator [20].

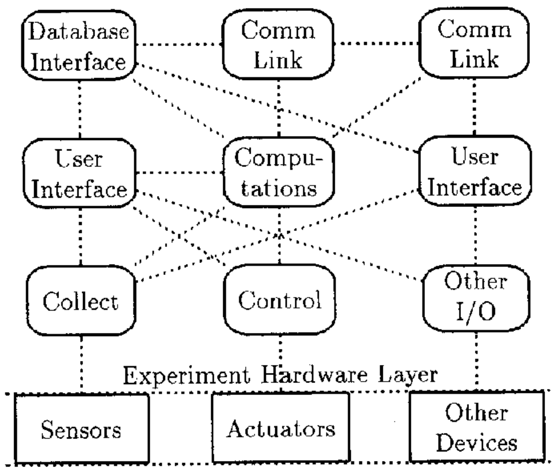

Fig. 13. Generic architecture of a distributed real-time system.

Multiple software units can be created to access various (maybe the same) sources and destinations of data and to exchange information among themselves. Any single unit can perform individual functions and communicate with every other unit.

This leads to the concept of a dynamic architecture, where the organization of components and their interconnections may change during execution [28, 32]. Adding or deleting new components should have no impact or minimal impact on the operation, in a sense that no degradation of functionality should occur due to such dynamic changes.

An interesting observation is that this type of architecture forms a tree, with a root at the experiment hardware layer. Only this root cannot change, everything else is flexible. Since only the root part is fixed, what happens above it is a matter of imagination of designers.

Communication links can operate individually or be lumped into a middleware layer [30], as in Fig. 14, with program units communicating partially or exclusively via this layer. Examples of both variants exist, in applications as different as satellite on-board embedded real-time systems [45] or large container terminal control systems in a big harbor [24], to name only a few published most recently.

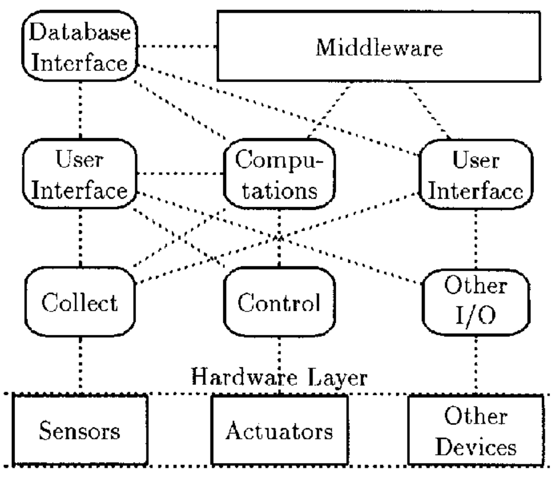

Fig. 14. Role of middleware in the architecture of a distributed real-time system.

The primary advantage of having such a flexibility is that a number of new components can be created and the architecture expanded during the run of an experiment or operation of a process. One such example is a dynamic GUI creation [31]. If new experiments are conceived which require including additional features to the GUI, or new characteristics are explored which need GUI reorganization, this can be done on-the-fly without jeopardizing the operation of an ongoing experiment or process.

6 Safety-Related Software Architecture

Another important category of real-time systems is that including mission critical or safety-related systems [3, 25, 26, 38]. Before discussing safety-related real-time architectures, it is worthwhile to take a closer look at the notion of safety and its relation to other critical system properties: reliability and security.

Considering critical system properties, we usually require guarantees on system behavior, requesting specifically that "nothing bad will happen" or that the risk of "something bad may happen" is low. The risk is usually analyzed in terms of potential hazards that are related to computer failures. If we ask the question, what is the risk, in terms of harm done to the environment or the computer system itself due to a computer (or software) failure, we may have the following answers:

1. Failure does not lead to severe consequences (high risk) to the environment or a computer system, nevertheless improving the failure rate is of principal concern (the notion of *reliability*).

2. Failure leads to severe consequences (high risk) to the environment, and later, maybe, to the computer system (the notion of *safety*).

3. Failure leads to severe consequences (high risk) to the computer system itself, and later, possibly, also to the environment (the notion of *security*).

In other words, reliability means minimizing undesired situations and their effects (to keep the system running), and safety and security mean preventing the environment or computer system, respectively, from undesired situations and their effects, because of the high risk involved (Fig. 15).

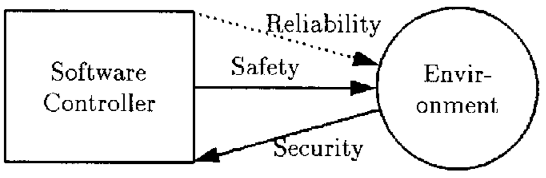

Fig. 15. Illustration of critical system properties.

Probably the most common mission-critical system every researcher is familiar with, although rarely considered in these categories, is a car. In addition to meeting the required level of performance (for example, regarding speed, acceleration, fuel consumption, etc.), an embedded microprocessor control software must meet several critical requirements, including:

- reliability, related to ignition control, cruise control, fuel gauge, odometer, etc.
- safety, related to air bag, seat belts control, anti-lock brakes, etc., and
- security, related to door locks, alarm, etc.

Stringent requirements on reaction times of air bag, anti-lock breaks, or alarms, make safety and security considerations a true real-time issue. In particular, a sporadic task handling the air bag release has one of the strictest requirements seen in practical applications.

The basic principle which should be observed when building software architectures for safety critical systems is "safety first". This comes from examples very common in practice, although not necessarily related to software, such as a lawn mower safety device, which is stopping operation immediately when an operator releases the handle. What is important to realize is that the actions of safety-related part of the system have precedence over the regular control procedure. This is confirmed for technologically more advanced systems, such as a nuclear reactor protection system (with a rod falling down into a reactor to absorb neutrons, in case of a danger).

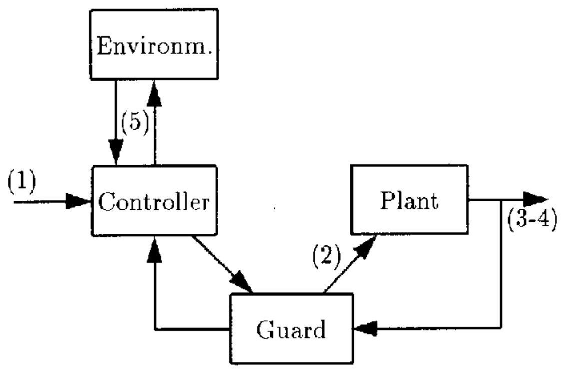

Fig. 16. Adding a safety guard to the structure of the control problem (notation as in Fig. 3).

Such observations suggest the use of a low-level construct, which acts as a guard detecting the danger first and then stopping the system and/or passing this information to a higher layer of control (Fig. 16). This is the role of a mower handle as well as that of the rods in a nuclear reactor protection system.

The assumption is to keep a guard as simple as possible, focusing only on the safety aspect. The safety component must first take care of all signals, before the controller. These factors limit the guard's basic functionality to reading the signals from the environment and determining whether they meet specifications, such as value ranges not exceeded, trends of variables within prescribed limits, validity of commands, etc. Thus the guard is filtering information passed to the control system. Particular algorithms what to do in case of detecting unsafe states depend on the application.

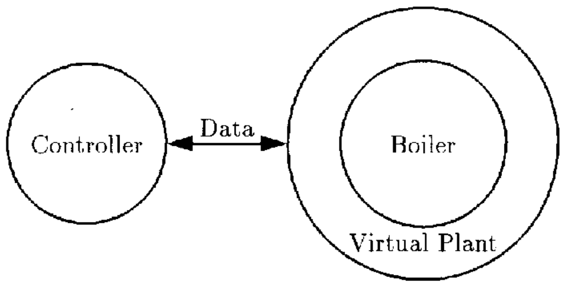

Fig. 17. Isolation provided by a guard as seen by the controller.

A guard conceived that way occupies virtual layers from the point of view of both the plant and the controller, by isolating the plant from the controller [49]. The isolation of a plant provided by a guard for the controller is illustrated in Fig. 17. The advantage of this situation is that the virtual plant is always in the safe state and the controller doesn't have to deal with errors in the plant at all.

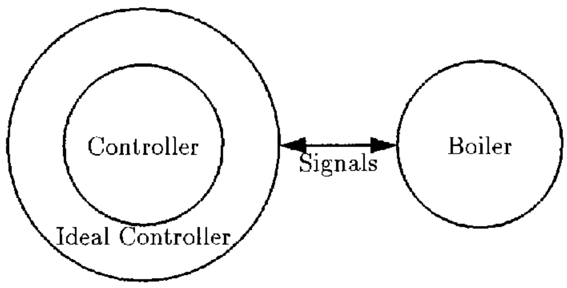

Fig. 18. Isolation provided by a guard as seen by the plant.

Because safety is a property that affects the environment (Fig. 15), the computer software must be watched too, not to contribute to the violation of safety. The advantage of the situation presented in Fig. 18, due to the use of a guard, is that the ideal controller takes care of controller errors and the plant is never affected by them. The only difficulty is when the guard itself is faulty.

7 Illustrative Example

As an example to illustrate how all these concepts worked in practice historically, we chose a water level control for a toilet tank [14]. An abstract view of the feedback control principle applied to a toilet tank is presented in Fig. 19.

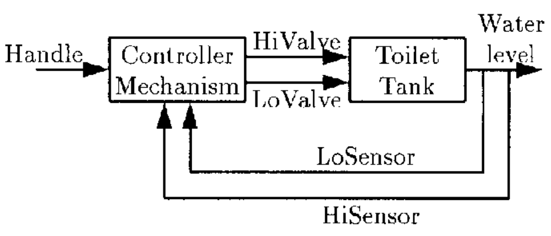

Fig. 19. Abstract view of the feedback principle in a toilet tank control.

In terms of a generic architecture (Fig. 3 & 4), it is the water level in the tank which is controlled, although via two different preset values:

- low level value, which is determined by a *LoSensor*, a float sensing whether all water left the tank
- high level value, which is determined by a *HiSensor*, another float sensing that water in the tank reached the highest allowed level.

Value of the water level is regulated via two actuators being the valves cutting flush water flow and input water flow, respectively:

- *LoValve* actuator is the valve controlling the flow of water leaving the tank
- *HiValve* actuator is the valve controlling the flow of water entering the tank.

In normal operation, the *LoSensor* float is responsible for detecting the water level at which the *LoValve* has to be closed. The *HiSensor*, in turn, is a float responsible for detecting the water level at which the *HiValve* has to be closed. Each sensor may actually be a part of a corresponding valve assembly, so it may be hard to separate them physically.

In addition to a plant interface via sensors and actuators, we also have a user interface, which is represented by a handle or button on the tank. A separation of a handle from the respective valve means in fact a careful user interface design.

It is interesting to see what is inside the toilet tank controller, in terms of its processing capabilities. Surprisingly, taking a closer look into this issue reveals a limited computational capability. Computation is in fact accomplished by the lever connecting the handle with the flush valve: pressing the handle causes lever to move and open the valve. This means clearly that the lever transforms an input signal into an output action and therefore performs analog computation, so in fact it is a computational device. Computation is based on mechanical principles and rather trivial but it is a computation.

By the same token, an earlier discussed water clock is a computational device, because it measures (computes) time. To achieve this, it performs arithmetic operation of summation by accumulating water. Being a hydraulic system, it also has storage, which changes its state when the tank is discharged. So does a toilet tank.

Why is it hard to observe computational capabilities or even believe that the computations take place? As pointed out in [5], analog computations suffer from the tyranny of impedance. Neither the toilet tank lever, nor the Watt's speed governor valve, considered as primitive amplifiers have a significant input impedance. Small input impedance of transformational (computational) elements means that a significant power has to be applied to the input and go through the computational device to cause any visible effects on the output. This is generally true for all non-electronic devices. High input impedance, that is, isolation of input from output was only achieved with the advent of electronic, in particular, digital technology.

Furthermore, since there are two control loops (one for each water level), this is a multivariable control system. Each loop performs its individual computational task and synchronizes with another loop via sequencing. The tank controller responds with an action closing the upper valve, only after the lower valve has been closed. This means we have a synchronization in a concurrent system. Moreover, even though there are no distributed computations *per se*, because the tank itself does not exchange any signals with external tanks, a toilet tank is a part of a very large distributed sanitation system, including sewege. The effects of distribution and interconnection are visible, if the sewege is clogged.

In this sense, the toilet tank control is also safety related. In particular, the design of the tank always includes provision for a failure mode operation. A fault tolerant element (such as a hollow tube in a tank) is used to drain excess of water directly into the toilet, to respond properly in case of an upper valve failure, which otherwise would cause water overflow and severe damage to property.

In summary, the toilet tank controller complies with all modern requirements for a control system, in terms of Fig. 4.

8 Real-Time Design Tools

Developing software architectures for contemporary real-time applications, such as those described in [20] is too complex to be done manually by a single individual. Therefore automatic tools are needed to assist in the development process. One such category of tools are architecture design languages.

From the developer's perspective, architecture design languages are similar to specification languages, in that they try to help formalize the design process. For this to be effective, however, a certain view of a design methodology should be followed. This view assumes that a complete design methodology must include the following three components [12, 27]:

- method, which is primarily a graphical notation applied to express important properties of an architecture
- techniques, that is, transformations which when applied to the notation allow for a successful completion of the development process (a set of such techniques including management aspects is often called a process)
- automatic software tools supporting the transformation techniques.

To this end, architecture design languages have not been very successful, mostly because of the lack of respective tool support and the lack of standardization. The importance of these two factors becomes more evident if we look at the success of design languages in other areas of computing. For example, VHDL is useful in hardware design, because it is standardized and supported by automatic software tools.

To be fully useful in the development of software architectures, however, the automatic tools have to provide support in the following four dimensions (Fig. 20):

- internal, related to all aspects of expressing real-time models via the specific notation and respective transformations
- horizontal, related to means of communication with other models and other tools (for example, via TCP/IP protocol suite)
- vertical, related to the next and previous phases of the development process, with respect to
  - support for code generation and prototyping, in particular, for specific programming languages and real-time kernels, and
  - support for design verification against the requirements (for example, to assure one-to-one correspondence of design components with the requirements)
- diagonal, related to the use of architectural models in different projects and different processes.

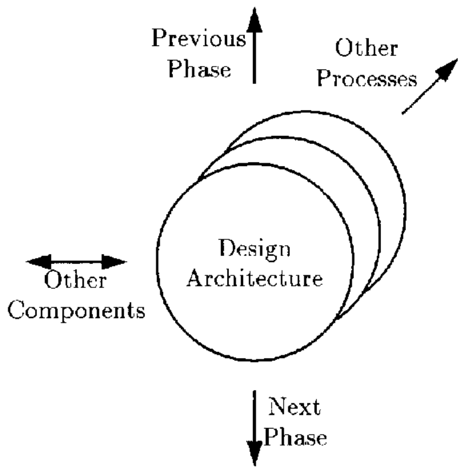

Fig. 20. Issues to consider when evaluating or selecting real-time design tools.

The advantage of this view, from the perspective of tool functionality, is that having the tools operational in all four dimensions, one gets the following properties addressed for the software architecture:

- completeness, correctness, and consistency in the internal dimension
- interoperability and connectivity in the horizontal dimension
- testability and traceability in the vertical dimensions
- reusability and portability in the diagonal dimension.

Several software tools with this concept in mind operate already in the commercial marketplace. For evaluation purposes, we focused

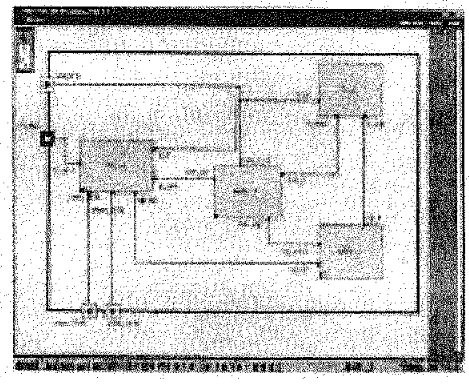

Fig. 21. Sample representation of a software architecture in a high-level design tool.

on a few tools, which have solid methodological foundations, based on object-oriented approaches: ROOM [37] and UML [16]. The detailed results of the comparative study are presented in [2]. In general, the study revealed several shortcomings of all tools with respect to the above mentioned 4 groups of criteria (dimensions).

In the internal dimension, the tools offer extensive capabilities to model both structure and behavior of distributed real-time software (Fig. 21 and 22). The most important capability of this sort is to create executable models based on statecharts [22] to study behavior (including animation). One disadvantage of the tools we evaluated is the lack of capability to perform quantitative timing analysis at the design level. None of the underlying OO methodologies, ROOM and UML, supports that either.

In the horizontal dimension, the tools usually allow simple communication of models with external objects at the TCP/IP level using sockets. This is possible for both communication with other models of the same type and totally independent program units running on a remote platform of almost any kind supporting TCP/IP protocol suite. In practice, this allows team development and testing of a complex distributed real-time system on different platforms, in different programming languages.

In the vertical dimension, the tools usually allow code generation for most real-time kernels. What they lack, however, is the tracing capabilities to go back from the code or design to the requirements. What some tools partially allow, for example, is incorporating into design the modifications made to a class source file, but for the most primitive operations only, which is very inadequate for complex real-time systems.

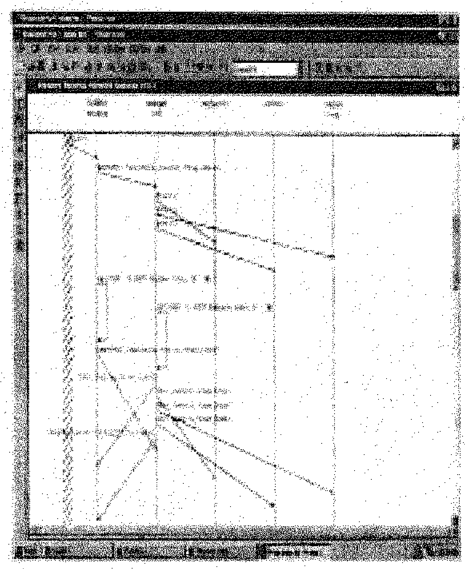

Fig. 22. Sample execution of a sequence diagram in a high-level design tool.

In the diagonal dimension, the tools have limited capabilities of importing models from other design tools but do not allow exchanging very comprehensive models among themselves. This may change in the future with convergence of the whole industry towards a UML notation.

9 Implementation Matters

At the implementation level, the design tools should allow timing and scheduling analysis, then correcting the design, and finaly code generation. With current tools, timing analysis is only possible after the code has been generated (although there is a new standard being developed [35], which may change it). To study their capabilities, we ran a case study of a complex distributed embedded simulation, composed of four comprehensive processes and an external interface (Fig. 23)

This comprehensive case study allowed us to test thoroughly all the capabilities of the tools involved with respect to the four-dimensional criteria listed in Section 8. When developing the basic structural components and defining their behaviors for distributed embedded simulation, it turned out that a much more rigid communication structure was necessary for the whole design. Traditionally, communication in distributed aplications is done via sockets or remote procedure calls (or equivalently, remote method invocation). This is very inadequate for distributed real-time applications.

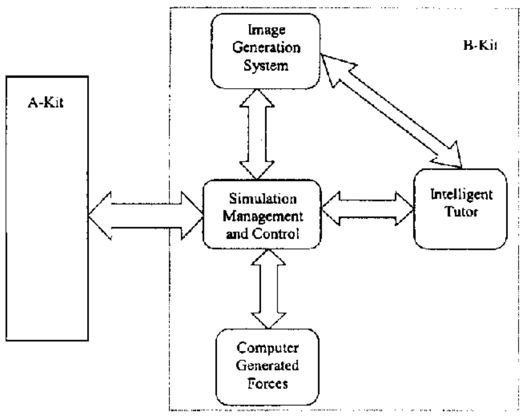

Fig. 23. System architecture of the distributed embedded simulation case study.

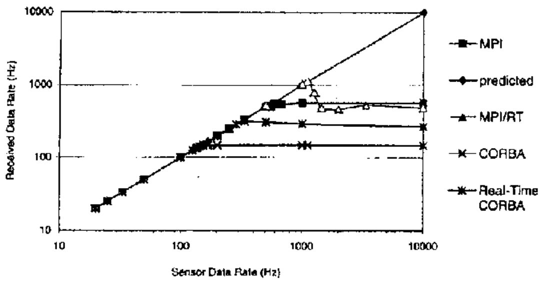

Fig. 25. Handling input load by MPI, MPI/RT, CORBA and RT-CORBA.

Therefore we included into this study two implementation level standards for distributed real-time communication: MPI/RT and Real-Time CORBA. A simple 5-task real-time benchmark program was designed and run under four stardards: MPI (mpich), CORBA (Visibroker), MPI/RT (from Mississippi State) and RT-CORBA/TAO (Washington Univ., St. Louis). Performance results for handling maximum communication load by these tools are presented in Fig. 24 [43] (for Sun Ultra 2 workstations running Solaris 2.6).

In summary, these architectural considerations led us to an important conclusion about the design and implementation sequence. This is that vendors of real-time design tools have to take into consideration not only the target platforms and respective real-time kernels but also the real-time implementation tools (such as MPI/RT and RT-CORBA) for the distributed target platforms. Including implementation related design patterns into design-level description will greatly simplify the process of designing real-time applications.

10 Conclusion

As one book states it [37], it is evident that *"The lack of a clear and accurate model of the actual system architecture has a severe impact on system understanding and is a prime contributor to system evolution and maintenance costs."* Therefore looking into the software architectures for real-time systems, that is, systems that must react within strictly defined timing constraints in response to external stimuli, should be of primary concern to software engineers and researchers.

In this paper, we tried to provide evidence that there is a clear template for real-time software architectures and design patterns, historically rooted in control engineering. From most likely the first engineered real-time device, a Greek water clock, to toilet tank controller, to steam engine speed governor, to air traffic control systems and distributed high-energy physics data acquisition systems, the same principles can be applied in designing real-time software architectures, forming a set of invariants that engineers can use further on in their design practice.

The most critical issue is to build a set of supporting software engineering tools to put these invariants into practice and assist designers in constructing real-time systems, both at the design and implementation levels. All this can be resolved as long as the software architects will follow the advice given over 2000 years ago by Vitruvius, in the first sentence of his monumental work: *The architect should be equipped with knowledge of many branches of study and varied kinds of learning, for it is by his judgement that all work done by the other arts is put to test.* [46]

References

[1] Al-Daraiseh A. et al., High-Level Tools Support for Integration Architecture in a Distributed Embedded Simulation Project, *Proc. CSMA2000 Conf. on Simulation Methods and Applications*, Orlando, Fla., 27-29 October 2000

[2] Al Mazid A.H.M., *Engineering Analysis of Object-Oriented Software Development Tools for Distributed Real-Time Systems*, M.Sc. Thesis, Univ. of Central Florida, Orlando, Fla., 2000

[3] Anderson E., J. van Katwijk, J. Zalewski, New Method of Improving Software Safety in Mission-Critical Real-Time Systems, *Proc. 1999 International System Safety Conference*, Orlando, Florida, August 16–21, 1999

[4] Atkinson C., C.W. McKay, A Generic Architecture for Distributed Non-Stop Mission and Safety Critical Systems, pp. 175-180, *Proc. 2nd IFAC*

Workshop on Safety and Reliability in Emerging Control Technologies, Pergamon, Oxford, 1996

[5] Auslander D.M., The Computer as Liberator: The Rise of Mechanical System Control, *Trans. of ASME: J. of Dynamic Systems, Measurement and Control*, Vol. 115, pp. 234–238, June 1993

[6] Baker T.P., G.M. Scallon, An Architecture for Real-Time Software Systems, *IEEE Software*, Vol. 3, No. 3, pp. 50-58, May 1986

[7] Bass L., P. Clements, R. Kazman, *Software Architecture in Practice*, Addison Wesley, Reading, Mass., 1998

[8] Bennett S., *Real-Time Computer Control: An Introduction*. Second Edition, Prentice Hall, Englewood Cliffs, NJ, 1994

[9] Benveniste A., K.J. Astrom, Meeting the Challenge of Computer Science in the Industrial Applications of Control, *Automatica*, Vol. 29, No. 5, pp. 1169–1175, 1993

[10] Boasson M., Control Systems Software, *IEEE Trans. on Automatic Control*, Vol. 38, No. 7, pp. 1094–1106, July 1993

[11] Buschmann F. et al., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons, New York, 1996

[12] Calvez J.P., *Embedded Real-Time Systems: A Specification and Design Methodology*, John Wiley and Sons, New York, 1993

[13] Clements P.C., Coming Attractions in Software Architecture, *Proc. Joint Workshop on Parallel and Distributed Real-Time Systems*, Geneva, Switzerland, 1–3 April 1997, pp. 2–9, IEEE Computer Society Press, 1997

[14] Coury B.G., Water Level Control for the Toilet Tank: A Historical Perspective, pp. 1179-1190, *The Control Handbook*, W.S. Levine (Ed.), CRC Press/IEEE Press, 1997

[15] Donohoe P. (Ed.), *Software Architecture. Proc. TC2 First Working IFIP Conference*, Kluwer, Boston, Mass., 1999

[16] Douglass B.P., *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patters*, Addison-Wesley, Reading, Mass., 1999

[17] Edwards C.H., D.E. Penney, *Elementary Differential Equations with Applications*, Prentice Hall, Englewood Cliffs, NJ, 1989

[18] Emery D.E., R.F. Hilliard II, T.B. Rice, Experiences Applying a Practical Architectural Method, *Proc. Ada-Europe '96*, A. Strohmeier (Ed.), Notreux, Switzerland, 10–14 June, 1996, Springer-Verlag, Berlin, 1996

[19] Fernández J.L. et al., A Case Study in Quantitative Evaluation of Real-Time Software Architectures, *Reliable Software Technologies*, L. Asplund (Ed.), Springer-Verlag, Berlin, 1998

[20] Gaspar K., B. Franek, J. Schwarz, Architecture of a Distributed Real-Time System to Control Large High-Energy Physics Experiments, *Parallel and Distributed Computing Practices*, Vol. 2, No. 1, March 1999

[21] Halang W. et al., Real-Time Computing Education: Responding to a Challenge of the Next Century, pp. 121–125, *Real-Time Programming 1997*, Pergamon, Oxford, 1997

[22] Harel D., M. Politi, *Modeling Reactive Systems with Statecharts*, McGraw-Hill, New York, 1999

[23] Hatley D.J., I.A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, New York, 1988

[24] van Katwijk J. et al., Software Development and Verification of Dynamic Real-Time Distributed Systems Based on the Radio Broadcast Paradigm, *Parallel and Distributed Computing Practices*, Vol. 2, No. 1, March 1999

[25] Krämer B., N. Völker, A Highly Dependable Computing Architecture for Safety-Critical Control Applications, *Real-Time Systems*, Vol. 13, pp. 237-251, 1997

[26] Leveson N., Software Safety in Computer Controlled Systems, *IEEE Computer*, Vol. 17, No. 2, pp. 48–55 , February 1984

[27] Ludewig J., *Zur Erstellung der Spezifikation von Prozessrechner-Software*. Doctoral Dissertation, Report KfK 3060, Kernforschungszentrum Karlruhe, 1981

[28] Magee J., J. Kramer, Dynamic Structure in Software Architectures, *ACM Software Engineering Notes*, Vol. 21, No. 6, pp. 3-14, November 1996

[29] Mayr O., *Zur Frühgeschichte der technischen Regelungen*, Oldenburg Verlag, München, 1969 (English translation: *The Origins of Feedback Control*, MIT Press, Cambridge, Mass., 1970)

[30] Muñoz C., J. Zalewski, Architecture and Performance of Java-Based Distributed Object Models, *Real-Time Systems Journal*, 1999 (submitted)

[31] Pedroza H., *GUI Builder for Real-Time Distributed Object Models*, M.Sc. Thesis, University of Central Florida, Orlando, Fla., 1999

[32] Polze A. et al., Real-Time Computing with Off-the-Shelf Components: The Case for CORBA, *Parallel and Distributed Computing Practices*, Vol. 2, No. 1, March 1999

[33] Pozesky M.T., M.K. Mann, The US Air Traffic Control System Architecture, *Proceedings of the IEEE*, Vol. 77, No. 11, pp. 1605-1617, November 1989

[34] Rechtin E., M.W. Maier, *The Art of Systems Architecting*, CRC Press, Boca Raton, Fla., 1997

[35] *Response to the OMG RFP for Schedulability, Performance, and Time.* Document Version 1.0, Object Management Group, 14 August 2000

[36] Schoch D.J., P.A. Laplante, A Real-Time Systems Context for the Framework for Information Systems Architecture, *IBM Systems Journal*, Vol. 34, No. 1, pp. 20–38, 1995

[37] Selic B., G. Gullekson, P.T. Ward, *Real-Time Object-Oriented Modeling*, John Wiley and Sons, New York, 1994

[38] Sha L., R. Rajkumar, M. Gagliardi, *A Software Architecture for Dependable and Evolvable Industrial Computing Systems*, Technical Report CMU/SEI-95-TR-005, Software Engineering Institute, Pittsburgh, Penn., July 1995

[39] Shaw M., Beyond Objects: A Software Design Paradigm Based on Process Control, *ACM Software Engineering Notes*, Vol. 20, No. 1, pp. 27-39, January 1995

[40] Shaw M., D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Englewood Cliffs, NJ, 1996

[41] Stankovic J., Misconceptions about Real-Time Computing, *IEEE Computer*, Vol. 21, No. 10, pp. 10-19, October 1988

[42] Stout T.M., T.J. Williams, Pioneering Work in the Field of Computer Process Control, *IEEE Annals of the History of Computing*, Vol. 17, No. 1, pp. 6–18, 1995

[43] Su S, *Benchmarking Distributed Real-Time Applications*, M.Sc. Thesis, University of Central Florida, Orlando, Fla., 2000

[44] Stuurman S., J. van Katwijk, Evaluation of Software Architectures for a Control System: A Case Study, pp. 157–171, *Proc. 2nd Int'l Conf on Coordination Languages and Models*, D. Garlan, D. LeMetayer (Eds.), Springer Verlag, Berlin, 1997

[45] Vardanega T., On the Distribution of Control Functions in New-Generation On-Board Embedded Real-Time Systems, *Parallel and Distributed Computing Practices*, Vol. 2, No. 1, March 1999

[46] Vitruvius Pollio, Marcus, *De architectura libri decem*, Rome, around 27–13 B.C. (Latest English translation: *Vitruvius: Ten Books of Architecture*, Cambridge University Press, New York, 1999)

[47] Witt B., F.T. Baker, E.W. Merritt, *Software Architecture and Design: Principles, Models and Methods*, Van Nostrand Reinhold, New York, 1994

[48] Zalewski J., Real-Time Data Acquisition in High-Energy Physics Experiments, pp. 112–115, *Proc. RTAW'93, IEEE Workshop on Real-Time Applications*, IEEE Computer Society Press, 1993

[49] Zalewski J., Boiler Water Controller Based on DARTS and EWICS Safety Model, pp. 223-231, *Software Safety: Everybody's Business*, D. Del Bel Belluz, H.C. Ratz (Eds.), Institute for Risk Research, Waterloo, Ont., 1994

[50] `http://www-ece.engr.ucf.edu/~jza/bld.jpg`