2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools

A design process for harware/software system co-design and its application to designing a reconfigurable FPGA

Félix Moreno[a], Ignacio López[c] and Ricardo Sanz[b]
[a]CEI - Centro de Electrónica Industrial, [b]ASLab - Autonomous Systems Laboratory
[a,b]Departamento de Automática, Ingeniería Electrónica e Informática Industrial
[c]Departamento de Ingeniería Energética y Fluidodinámica
Universidad Politécnica de Madrid (UPM)

Abstract – This paper is going to address the topic of hardware/software systems co-design. The paper will develop two points of view. First, it provides a system-theoretical layout on the problem of designing hardware-software systems. This layout will enable the designer to proceed systematically in optimizing the tradeoff between the desired functionality, available resources and operating conditions. Second, the paper will describe an application of some of the theoretical principles to the design of an embedded automotive system built on a low-cost FPGA.

Keywords: Hardware/software co-desing, uncertainty handling, cognitive architecture, FPGA, blackboard.

I. Introduction

The need for optimization of systems in resources and performance implies a drive for hardware-software co-design in order to achieve a strong coupling between algorithms and functions on one side (software) and platform on the other (hardware). This is required in order to satisfy strong real-time specifications, advanced functionalities or cost limitations, in many cases simultaneously.

Many advanced applications which are being developed today using large computational resources are intended to be embedded in the near future. Some examples of these are space applications, automatic driving, collision warning and active safety systems and interfaces which interact with people in natural language and mood. The majority of these applications rely on highly resource-consuming artificial intelligence techniques, which enable them to react adequately to the complex environments in which they have to operate. Apart from the nature and complexity of the applications themselves, many must show a high degree of dependability for safety-critical purposes [5][9].

Embedding this kind of applications with minimal resource/performance ratio requires hardware/software co-design methodologies [9]. Dependability, cost and performance must be optimized. They rely on several critical design aspects: a good functional specification, a good system model, a good environment model, and an streamlined design process.

This paper will develop the last point by describing a generic design process for embedded intelligence applications. The main contribution of the paper will be to identify the decisive factors in the design process which the engineer must evaluate, and an overview of the tradeoffs and constraints between them. This vision will then be exemplified with a commented description of a concrete design process realisation. The objective was to build a prototype for an embedded system for vehicles, to detect traffic signals in real world environments. The intelligent system was built on a low-cost FPGA by embedding a high level, blackboard-based intelligent architecture which had been originally used for other applications like mobile robot control, medical systems and social interaction(BB1 architecture from Stanford KSL). Only design-procedural aspects relative to hardware/software co-design will be commented on this paper.

II. A perspective on hardware/software systems co-design

In brief, a methodology for hardware / software systems co-design must achieve an efficient tradeoff between three factors: *desired functionality*, *system* and *operating environment* [12]. They have to be understood in the wide sense. A desired functionality is the purpose for which the system is built in combination with any constraint or additional specification on how it must carry out this purpose. The system is the combination between a set of hardware resources which we will call *platform* and a set of software resources which we shall call *functions*. In these terms, the operation of the system is its set of functions executing over its platform. The *environment* is everything surrounding the system that may affect it. For the designer, the environment is not important in itself, but only to the extent given by the system and the desired functionality [8].

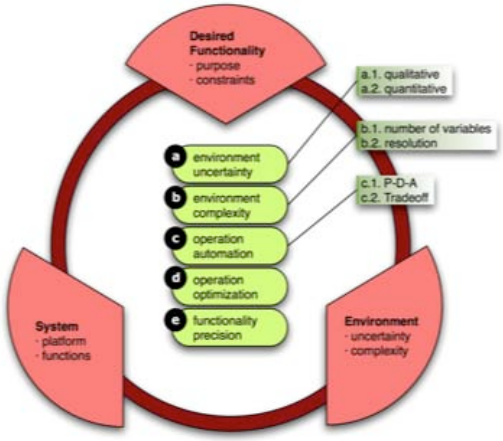

Figure 1. Some of the aspects intervening in hardware / software systems co-design.

978-0-7695-4171-6/10 $26.00 © 2010 IEEE
DOI 10.1109/DSD.2010.43

We shall assume that a design process starts with a certain specification of the desired functionality, and of the system resources to use. The specification of the desired functionality may include some description of the environment where the system will be used. The objective is to find the architecture and functions capable of achieving the desired functionality with the given resources. We shall assume a secondary objective which is to discard any resources which are not strictly needed.

Thus, a process of hw/sw system co-design will consist in a number of iterations leading to an efficient match between the three factors: functionality, system and environment. By efficient we mean first, that it satisfizes or improves the specifications given to the designer in functionality and cost; second, that it is physically realizable with the selected platform and functions.

Many constraints, difficulties and problems may arise during the design of a hardware/software system. The uncertainty of the environment is the fact of ignoring the precise nature and intensity of the behaviour of the environment. In other words: ignoring exactly *what* is going to happen and ignoring *how intense* it is going to be. We shall call the first *qualitative uncertainty* and the second *quantitative uncertainty*. Building systems capable of operating in highly uncertain environments may require many resources. In general, designing for qualitative uncertainty implies building sophisticated models with many variables or costly competences for cognitive adaptation. Designing for high quantitative uncertainty requires a certain degree of precision in measurements and actuators and the existence of robust control mechanisms. Normally, real environments present both types of uncertainty. Controlled environments, such as industries or laboratories, generally eliminate qualitative uncertainty and minimize intensive uncertainty to an extent that it can be easily compensated for by simple controllers –e.g. by conventional PID controllers.

Valuating the type and intensity of uncertainty associated to a particular design problem is critical. Depending on the level of uncertainty more or less variables will be required and more or less precise they must be for modelling the environment that the system will be operating in, and the complexity of the algorithms which it will implement. Thus, it determines an order of magnitude for resource requirements. A correct valuation of uncertainty is that which uses the minimal number of variables with the minimal resolution possible to correctly represent and execute the algorithms that allow the system to reach its objective. In the majority of cases it will result from a number of iterations. The experience of the designer is decisive.

Uncertainty defines the context for a design problem. The first proper stage of a design process, following a correct evaluation of uncertainty, is the design of the system architecture. In hw/sw systems, the architecture includes both parts. An adequate architecture will specify a consistent structure of subsystems, interfaces and integration mechanisms. It will establish an adequate splitting of system functions between hardware and software, and specify the requirements for the coordination mechanisms between both.

An important feature of a system's architecture is its *modularity*. By this we mean "the degree to which a system can be separated and recombined" [13]. An adequate specification of subsystems and interfaces may increase or prevent modularity. The basic technique in which modularity is based is in designing interfaces between the parts of a system, so that an interface groups all the channels of interaction of one part with the rest. This provides a well defined structure of communication between parts, which allows tracing and controlling the flux of inputs and outputs. More evolved techniques, based in this principle, lead to operating systems and middleware [16]. Although these might be excessively resource consuming for many embedded systems, they may be useful in some applications, especially if they are distributed and under real-time or synchronization requirements. This is the case of automotive applications, based in a variably wide collection of communicating embedded systems across the vehicle. A special operating system standard and an architectural specification have been developed [2][11].

Modularity is desired for a number of reasons. First, from the point of view of the developer, it allows developing, modifying and upgrading each module separately from the rest, provided it meets the specification of its interfaces. Second, from the point of view of system operation, it makes the system easier to coordinate, it opens the possibility of the system using or discarding its own modules as part of its own operation and it enables implementing fault-tolerance mechanisms (it makes fault detection easier and is required for damage confinement) among other advantages. It may lead to greater resource requirements, so in the case of strong limitations it may have to be discarded. Otherwise, the overall system gain in efficiency and coordination may compensate for greater resource consumption in the modules.

Another feature of the system architecture directly related to its mode of operation is the degree of automation of the functions. Architectures for design problems involving low levels of complexity may be completely automated. In more complex ones, especially if the complexity derives from qualitative uncertainty, only some functions may be fully automatic, while others may require some degree of inference or learning. In hardware/software co-design for embedded systems, automating as many functions as possible is interesting in general. In many cases, it is equivalent to deciding whether to make functions memory- or processor-based.

In general, automatic functions can be implemented memory-based, as in the well-known case of automotive engine control, where the control law is stored in a ROM memory and consulted in run time by *lookup-table* [3]. This reduces the number of calculations to carry out to a minimum increasing dependability and freeing the processors for executing other functions. Memory-based implementations can be tested more easily and may be deployed over lower-cost hardware. As to performance, they are usually faster than processor-based

implementations, for they entail only basic arithmetic operations (comparison, interpolation) and memory access.

Processor-based implementations calculate in run time. This might be associated also to large memory consumption for operands and intermediate results. This might be highly resource-consuming and not always affordable in embedded systems. They might compromise system dependability due to essentially two reasons: they might be affected by environmental conditions (temperature, EMI) and they might require implementing efficient run time support and scheduling algorithms for sharing the processor and managing their execution (halting, prioritizing, aborting, etc.) [16].

The previous points are architectural, but may also apply to detailed design. Detailed design establishes how to build the structure of system parts, associated functions, algorithms and interfaces specified by the architecture. This stands for deciding clock frequencies, bandwidths, designing pipeline structures, mechanisms of memory sharing, parallel and serial processes, etc. We are all conscious that a same algorithm can be programmed or coded in different ways, not all being equally resource consuming, fast and efficient. As a result from this, a same architecture can be made optimal or unviable.

In conclusion, a design process must integrate desired functionality, system and environment. A deficient design or characterization of any of these three factors will drive the resulting design away from the optimal in cost, resource use or performance. It may even turn it unviable. A systematic application of problem analysis and design criteria for optimization throughout the design process is required. This in turn requires the designer to know the system-level implications of the design solutions being considered. For this, it is useful to think in terms of resource/performance ratio, modularity and operation automation, and how they can be reflected on architecture and detailed design.

III. A PROCESS FOR HARDWARE/SOFTWARE SYSTEMS CO-DESIGN

In this section we are going to propose an iterative process for hardware/software systems co-design, summarized in the figure. It is based on system theory [8], software engineering theory [13][15], application development techniques [3] and the past trajectory of the research team in embedded system design [1][10].

The first steps of the process (0-4) will have more importance at the evaluation and basic design stages of an engineering project. The later steps (4-5) at the final stages concerning detailed design. A final step (8) called *refine functionality specification* has been included here explicitly, because it is understood that it may be significant in the design of embedded systems for mass production.

The process consists of five steps that represent a conventional design sequence, indicated by thin arrows: 0, 1, 4, 5 and 8. The proposed process for hardware/software co-design is indicated by block arrows, and leads to three kinds of loops labelled A, B and C, and to intermediate steps 2, 3, 6 and 7.

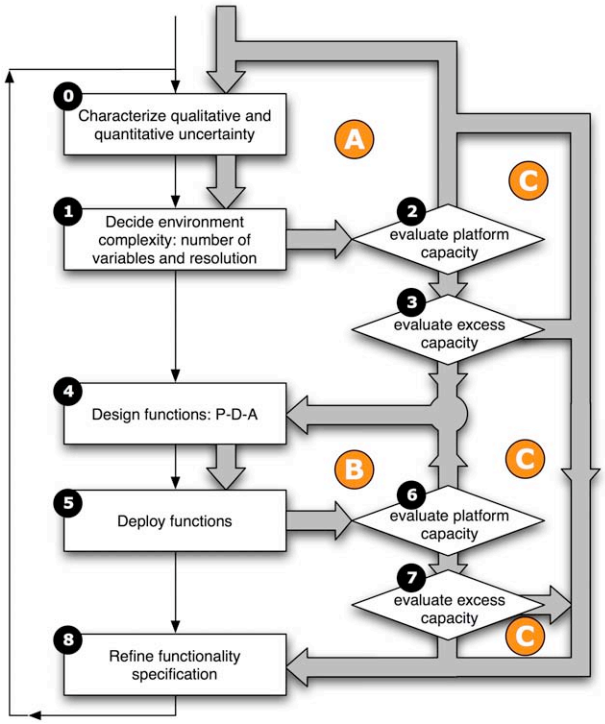

Figure 2. Process for hardware/software co-design

Steps 0 and 1 stand for evaluating and quantizing the complexity of the problem. Fixing the number of variables with which the system is going to operate and their resolution. Step 4 stands for designing the system architecture, subsystems and interfaces, algorithms, and deciding which are to be implemented in software and which in hardware. Step 5 consists in the detailed design of subsystems and algorithms. It may include partial testing, prototyping and redesign. Step 8 stands for improving the current functional specification according to the specific characteristics of the design learnt in the previous phases.

We may realize that most software design is concentrated on steps 0,1 and 4. Most hardware design on step 5. In the design of standard hardware-based systems, step 5 might be limited to assembling pieces, compiling and checking the software generated in steps 0-4. This is the case of many functional prototypes which are implemented in desktop computers and tested before being deployed on chip. In embedded systems, a strong interaction must exist between hardware and software in order to maximize dependability and resource use. They must be adequately coordinated (for an overview on real-time system topics see).

The proposed block-arrow process structures the sequantiation of design phases for obtaining the strong software-hardware coupling required in embedded systems. Let us describe the A, B and C loops:

- **Loop A.** It focuses on the contextual aspects of the design, basically evaluating the minimal requirements for modelling the environment in which the system will be operating, both dynamically and statically. After some iteration, an experienced designer may also be able to grossly estimate the order of magnitude for the

resources needed for operating with these models (basically processor speed, bandwidth and memory). If the proposed platform does not match the results of the analysis, either the design must be terminated or its functionality redefined: its scope reduced or the target platform changed for a more powerful one. The proposed platform should exceed the requirements moderately to allow for additional needs that could eventually appear in future steps. This excess might be required if the system design is intended to be future-proof, modular or scalable. A massive excess may indicate resource waste, and should lead to changing the target platform in step 8.

- **Loop B.** This loop contains the design process as such. The basic design (4) comprises architecture, subsystem, interfaces, integration mechanisms and algorithms. It also includes deciding which algorithms are to be implemented in software and which in hardware, and designing the mechanisms to ensure coordination (in real-time system language, *run-time support system*, *RTSS*). Detailed design (5) comes from developing the previous into software programs and specific hardware designs for the target platform (frequently VHDL code). The difficulties and limitations encountered in this step may lead to adapting the basic design of (4). Step 6 must be carried out systematically in order to ensure the feasibility of the architectural requirements derived from (4) and the designs of (5) in combination. This last aspect is important: The algorithmic dependence between functions is established by the architecture generated in (4). If cost, size or weight requirements are strong, as in the case of embedded systems for mass production, it will not be possible to implement the several different functionalities independently and resource sharing will be required. This will cause extra dependence between functions apart from the algorithmic relations structured by the interfaces specified in the architecture. This dependence might require extra resource management functions (resource allocation, prioritization, synchronization), which must be included in the loop.
- **Loop C.** This loop includes all ways leading to improving the functionality specification. It must concentrate on:
  - 8.1. Fixing architectural and functional features: scalable/enhanceable/fixed system
  - 8.2. Limiting the scope of the functionality
  - 8.3. Eliminating ambiguities and providing measurable requirements
  - 8.4. Adjusting the platform to the above: adding/eliminating hardware resources to the design

IV. An application example: Designing a vehicle-embedded system for identifying traffic signals in real environments

This section describes a real design which was prototyped for testing different functionalities and technologies being investigated at the time. A detailed description of the architectural design can be found in [6] and of the final implementation, including hardware synthesis results, in [7]. The project was developed within a line of research on scaling high-level intelligent architectures into embedded systems for commercial applications.

The purpose of the system was detecting traffic signals and warning the driver in real time and urban driving conditions. The system must be enhanceable to pedestrian detection and other possible applications in the future. The system must minimize costs and size in order to be mass-produced in cars.

In engineering terms, this problem implied moderate levels of both qualitative and quantitative uncertainty, in that the possible signals to appear belonged to a finite set, and that they could appear partially occluded, shaded or rotated. On the other side, the signals would most probably appear in fixed regions of the windscreen. There also existed very limitating resource constraints, which we pushed to the limit.

The platform consisted on a camera installed on the dashboard facing the road, a FPGA-based board for real-time image processing, and a very limited human-machine interface (HMI) consisting of a 3-led display. In order to minimize the cost, the camera was a low-cost PAL device and the FPGA was an Altera Cyclone.

The requirements for scalability and enhanceability were met by adapting the BB1 high-level blackboard architecture [4] that had been used for autonomous robots, social interaction with humans and automatic medical monitoring.

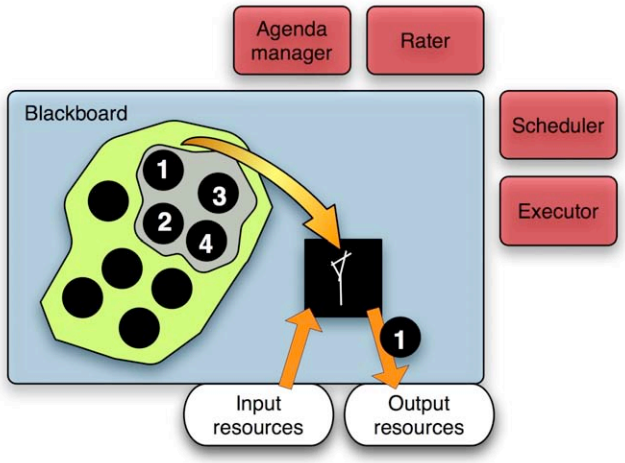

Figure 3. Blackboard architecture

Very briefly, a blackboard architecture consists of four groups of resources: input channel, output channel, blackboard and agents (see figure). Although in the original BB1 the input channel was a complex part, its operation is conceptually similar to the combination of *sensor-conditioner-driver-interface*. Analogously with the output channel and the *driver-actuator* set respectively. The blackboard contains knowledge in the form of production rules represented as black circles in the figure. The agents analyze the input to the blackboard produced by the input

resources (black and white images in our case) using the knowledge stored in the blackboard. Each rule is evaluated against the input and grouped into a set if it could be applied to it (shaded in the figure), otherwise discarded. This is done by the *agenda manager*. The *rater* then assigns a priority among the applicable rules as shown in the figure. The highest priority rule is selected and adapted to be executed by the *scheduler* (resource allocation, device configuration, etc.) and executed through the output resources, under the control of the *executor*. In our case there will be only three possible actions: activating the corresponding light in the 3-led display depending on the type of traffic signal detected.

A. Input resources

PAL image digitizing into 720x576 pixel frames at 25 fps, median and adapted sobel filtering. The results were black and white images of borders. Only a 60x45 region of the frames was analyzed for traffic signals, *Region of Interest, RoI*. This region is the most probable for a signal to appear in. Digitization, processing and RoI extraction were all implemented in the FPGA.

B. Blackboard

In the conventional BB1 architecture, different types of knowledge coexist. The unit of knowledge is an *object*, represented by complex data structures. We simplified this by making our architecure behave as if executing production rules: "if-then" rules. However, storing these rules explicitly as data structures as in the original architecture was impossible given the resource restrictions. Additional problems related to quantitative uncertainty also made this impossible. We managed to simplify drastically process of identificating signals:

- *Theoretical reduction of the problem:* In real environments, there exists a moderate quantitative uncertainty: every new image of a same traffic signal might show differences in lighting, occlusion, angle and position. The range of variability is so broad that the detection is very difficult to automate. Full computer-based systems solve this problem by pattern matching and other artificial vision techniques out of the computational range of our platform. We used the concept of *singularity*: a characteristic feature of the object to be perceived, traffic signals in our case. Instead of attempting to identify the whole object, we deduced its presence by detecting singularities. We considered three: a right-angle corner, an acute corner and a curved segment. Once the singularities are detected in the RoI, deducing to which signal they correspond can be automated. We built three units of knowledge, one for each singularity, which we called *knowledge units*.
- *Implementation:* The theoretical reduction achieved by the singularities was not enough to eliminate all quantitative uncertainty. Although simpler than analyzing for skewed triangles, squares and circles, singularities could appear also skewed, rotated or misplaced. We decided to overcome this by using a small neural network for detecting each singularity. The capacity of the FPGA, however, was not sufficient to hold the three (implementation results show a 54% use of the Altera Cyclone's M4Ks for implementing a single network). We decided to share the FPGA resources between them: we decided to run the three networks sequentially. This was possible because the execution time of the hardware neural network design that we produced was short enough (in the order of 70 μs to process a RoI) not to cause conflict with real time constraints (1/25fps = 40 ms). This entailed the problem of reconfiguring the FPGA on line. The strategy which was adopted was to implement a *generic neural network*, GNN, in the FPGA, whose weights were built as registers which could be loaded in run time. The three sets of weights, one for each network, were stored in memory. They were loaded into the GNN and run one after the other for each image.

In summary, the camera delivered PAL video to the FPGA board which performed digitization, filtering and RoI extraction (input channel). The results of this were 60x25 images delivered to the blackboard at 40 ms intervals. An incoming image was analyzed for the presence of three singularities: right-angle corners, acute angles and circular segments. This was carried out in sequence by a single hardware neural network, reprogrammed in sequence with the corresponding set of weights for detecting each singularity. The result was interpreted with a combinational circuit in the FPGA acting as rater, scheduler and executor, and the result shown by lighting the corresponding diode in the 3-led display. The system had capacity for discriminating round, square and triangular traffic signals.

V. DISCUSSION ON THE PROCESS OF HARDWARE/SOFTWARE CO-DESIGN

Most of the aspects and techniques described in the initial sections of this paper were exercised in this project. The extremely demanding functionality and limited resources were a design challenge throughout. Naturally, the final solution was reached after considering many intermediate designs, prototypes, tests or, in other words, many iterations of the proposed hardware/software co-design process. Several interesting aspects must be remarked:

A. Enhanceability, scalability and qualitative uncertainty

The three requirements can be satisfied with the blackboard architecture. It allows separating knowledge from agents. Because of this, making the system capable of detecting one type of traffic signal equals to giving it the appropriate knowledge (one or more rules). For the same reason, making the system capable of detecting pedestrians in the future will consist in adding rules.

Although this is basically true, our implementation ran out of resources for a proper, modular implementation of rater, scheduler and executor. Adding new knowledge would therefore require adapting the version referred to

here, and would probably imply using more powerful hardware.

The problem of qualitative uncertainty is solved by the design of the architecture itself. Due to the separation of knowledge and agents, and to the cycle of operation of the architecture, the system will adapt automatically to any new signal appearing in view.

In strict architectural terms, choosing a blackboard approach provides scalability. Its modular design separating input, output, blackboard and agents makes possible to update or enhance one of them separately from the rest, or to implement it apart with dedicated hardware, only by providing minimal additional mechanisms for integrating all components. During the project, the idea of implementing the input resources on a separate FPGA was considered and left for future work.

Regarding modularity, it is worth mentioning that the VHDL design produced for the GNN was highly modular. It was based on self-controlled perceptron layer modules which allowed coupling any number of layers without global control. Although the GNN in this system was a single layer perceptron, the design was prepared for future, more complex applications.

B. Quantitative uncertainty

This was a major problem throughout the project. Three design solutions were made in order to minimize it:

- Reducing the resolution of the incoming resources by noise-filtering, extracting borders and limiting the image to black and white. The increase in complexity of richer representations and the processing algorithms they would have entailed made the design unviable.
- Conceptual simplification of the problem: Instead of attempting to detect shapes, simpler patterns, singularities, were detected, and the presence of shapes deduced from them.
- Neural networks: The robustness of neural networks against rotation, scaling, etc. was still required in spite of the above simplifications.

C. Limited resources

Many design decisions were made in order to adequate the system to the Altera FPGA. The major decision was to reduce the desired functionality from full traffic signal detection to discriminating three types.

In the design, the most relevant ones are the following:

- Analyzing a Region of Interest within the full frame instead of the whole frame.
- Sharing the FPGA between the three networks. This required extra control mechanisms: registers, buses and flags.
- Limiting the characterization of the signal types to three singularities: Alhough tests show good performance results, a larger number of singularities would be needed for commercial-level dependability (including night detection and a wide range of weather conditions). The pass from detecting signal-types to detecting specific signals would also require a larger number.
- Collapsing rater, scheduler and executor to combinational logic. Logically, as it has been commented, this reduces system modularity and thus limits scalability and enhanceability.
- Eliminating other functionalities: The original BB1 architecture used automatic learning capabilities. They are based on complex schemes of data representation, intensive processing and memory usage.

D. Time

We assumed the restriction that the system should produce a detection on a frame-by-frame basis, which gave a 40 ms deadline. As a result of all simplifications and the optimized hardware implementation of the GNN, it was possible to execute the cycle of image preprocessing, singularity detection, signal type deduction and output in times near to 30 ms using a 100 MHz clock.

This performance rate is critically dependent on the detailed design of the parts. The same cycle of operation was tested with earlier versions of the preprocessing architecture and the GNN, with the result that the 40 ms deadline was not met.

VI. CONCLUSIONS

In order to produce an efficient design, a conventional design process must be complemented with additional criteria. The objective of these criteria is to intervene in making design decisions, by referring them to complexity, resource/performance ratio, modularity and operation automation. It must define a sequence of phases which allows the designer to apply them systematically.

In this line, the hardware/software co-design process proposed in this paper is a conceptual framework with which to systematize the design process requiring high integration between hardware and software. It structures and defines the relative importance of many ideas related to system design which are seldom unified in a single methodology.

The design described in this paper is a good example of most of the ideas included in the proposed co-design process. It includes strong resource limitations, real time constraints, qualitative and quantitative uncertainty and the need for redefining the desired functionality of the original project. The exact places in the design process in which complexity, uncertainty, architecture, modularity, resources and optimization appear can be identified.

REFERENCES

[1] J. Alarcón, R. Salvador, F. Moreno, P. Cobos, and I. López, “A new real-time hardware architecture for road line tracking using a particle filter,” in Proceedings of the 32nd Annual Conference of the IEEE Industrial Electronics Society, IECON’06, Paris, France, pp. 736–741, IEEE Industrial Electronics Society, November 2006.

[2] A. GbR, “Specification of the virtual functional bus,” Standard V1.0.2 R3.1 Rev 0001, Autosar, August 2008.

[3] *Bosch Automotive Handbook*. ISBN 0837615402, Robert Bosch GmbH, 7th ed., November 2007.

[4] B. Hayes-Roth, "An architecture for adaptive intelligent systems," *Artificial Intelligence*, vol. 72, no. 1-2, pp. 329–365, 1995. http://citeseer.nj.nec.com/hayes-roth95architecture.html.

[5] D.-S. Huang, L. Heutte, and M. Loog, eds., *Advanced Intelligent Computing Theories and Applications. With Aspects of Contemporary Intelligent Computing Techniques*. No. ISBN 978-3-540-74281-4 in Communications in Computer and Information Science 2, Springer, 2007.

[6] I. López, R. Salvador, J. Alarcón, and F. Moreno, "Architectural design for a low-cost fpga-based traffic signal detection system in vehicles," in Proceedings of SPIE Europe Microtechnologies for the New Millennium (SPIE, ed.), ISBN 9780819467188, 2007.

[7] I. López, R. Sanz, F. Moreno, R. Salvador, and J. Alarcón, "From cognitive architectures to hardware: a low cost fpga-based design experience," in Proceedings of the IEEE International Symposium on Intelligent Signal Processing, ISBN 1-4244-0830-X, pp. 499–504, IEEE, 2007.

[8] G. J. Klir, *An Approach to General Systems Theory*. Van Nostrand Reinhold, 1969.

[9] S. Lu and W. A. Halang, *Contributions to Ubiquitous Computing*, vol. 42/2007, ch. A UML Profile to Model Safety-Critical Embedded Real-Time Control Systems, pp. 197–218. Springer, 2007.

[10] F. Moreno, F. Aparicio, W. Hernández, and J. Páez, "A low–cost real–time fpga solution for driver drowsiness detection," in Proceedings of the 29th Annual Conference, IEEE Industrial Electronics Society, IECON'03. Virginia (USA), IEEE Industrial Electronics Society, 2003.

[11] *OSEK/VDX Operating System Specification 2.2.3*. OSEK, February 2005.

[12] R. Sanz, F. Matìa, and S. Galán, "Fridges, elephants, and the meaning of autonomy and intelligence," in *Proceedings of the 15th IEEE lntemational Symposium on Intelligent Control (ISIC 2000)*, pp. 217–222, IEEE, July 2000.

[13] R. Sanz, I. Alarcón, M. Segarra, J.A. Clavijo and A. de Antonio. Progressive Domain focalization in Intelligent Control Systems. *Control Engineering Practice*, Volume 7, Issue 5, May 1999, Pages 665-671, 1999.

[14] M. A. Schilling and C. Paparone, "Modularity: An application of general systems theory to military force development," *Defense Acquisition Review Journal*, pp. 279–293, 2005.

[15] I. Sommerville, *Software Engineering*. ISBN 0321313798, Addison Wesly, 2008.

[16] X. Wang, C. Lu, and C. Gill, "Fcs/norb: A feedback control real-time scheduling service for embedded orb middleware," *Microprocessors and Microsystems*, vol. 32, no. 8, pp. 413 – 424, 2008.