Automotive engine/chassis control and development systems: quick state-of-the-art overview focused on a "clean-slate approach" for innovative easy-handling methods

Pedro Kulzer, Bernardo Cunha

Resumo - O presente artigo é iniciado com a identificação do sistema automóvel motor/chassis, alvo do estado-da-arte das ferramentas de desenvolvimento e controlo apresentadas logo de seguida. Aqui mostra-se a necessidade inerente de mecanismos easy-handling que permitam conseguir-se dar conta da gama sempre crescente de detalhes programáticos e operacionais das respectivas centralinas (ECUs). Esta complexidade surge, em grande parte, devido à elevada opacidade do caminho entre a interface do utilizador e a plataforma de hardware. Finalmente, são feitas algumas considerações sobre melhoramentos que potencialmente alteram significativamente o modo de programar, prototipar, simular, depurar e verificar sistemas automóveis. No final são ainda apresentados alguns detalhes elucidativos destas melhorias, a partir dum projecto a decorrer e baseado num "clean-slate approach". É mostrado que encurtando drasticamente o caminho entre interface e hardware, surgem possibilidades interessantes como o "Live-Prototype" que se resume a uma espécie de Programação 100% Interactiva. O paper [1] é uma excelente introdução para se compreender melhor as preocupações aqui apresentadas.

Abstract - This paper starts by identifying the automotive engine/chassis system, which is the target of the state-of-the-art development and control tools presented right after. The underlying need for easy-handling mechanisms to harness the overwhelming and ever-growing range of programming and operational details of the corresponding electronic control units (ECUs), is addressed. This complexity is mainly caused by the highly opaque path between the user-interface and the hardware platform. Finally, some considerations on enhancements are made, which potentially provide significant changes in the way automotive systems are programmed, prototyped, simulated, debugged and verified. At the end, some elucidative details of an ongoing "clean-slate approach" project are disclosed. It is shown that by shortening the path between interface and hardware, interesting possibilities arise, such as "Live-prototyping", which is a sort of 100% Interactive Programming. The paper [1] is an excellent introductory reading to better understand the herein presented worries and considerations.

I. INTRODUCTION

Current automotive control and development systems used in the most renowned car and also in the car-component manufacturers are complex systems which still use classical imperative textual programming techniques and standard handling methods. The corresponding tools, although being state-of-the-art themselves, center upon historically layered-grown complex software tool-chains and on correspondingly inflexible hardware platforms. Very sophisticated visual tools have been emerging, which allow to optimize development efforts [2] [3] at the top-most automotive control-functionalities design. But these systems still rely, at some internal level or layer of their assembly, on classical mechanisms such as compilers and assemblers. These lower tools in turn use standard textual imperative languages such as mainly "C" for building machine-code for downloading onto standard and rigid micro-controller based hardware. Although the top-most interfaces are now visual, even state-of-the-art tools still generate "C" code for use by lower-level components [4].

Most of the engine/chassis management functionalities rely on standard signal-processing and data-flow methods. Modern automotive and industry-related software tools take advantage of that fact and concentrate on visual data-flow programming paradigms [5] [7] [8] [11], therefore avoiding the hassle of classical low-level textual control-flow programming at top-level. Even standard tools which use of control-flow programming at user-interface level use internal data-flow mechanisms such as time-raster calls with well-defined data-oriented execution sequences, fully mimicking this data-flow centered nature.

All of the hardware systems rely on a closed control-loop comprising the electronic control unit (ECU) itself, which contains the control-algorithms (software), and finally the plant to be controlled (vehicle). Control- and sensor-signals represent the data-flow between the ECU and the car components (Fig. 1). Unfortunately, the hardware solutions did not follow the automotive data-flow nature as closely as the visual tools did. In fact, they still strongly reflect control-flow structures, therefore producing some very severe conceptual mismatch between software and hardware, thereby forcing visual tools to produce control-flow classical "C" code under their hood.

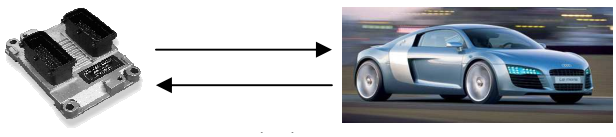

Fig. 1 - ECU and car closed-loop control with in/out signals

This paper will essentially addresses software development tools used both in series-car and motorsport areas, especially the complexity that arises from the conceptual mismatch between development user-interfaces and the corresponding hardware target platforms. This complexity is thus essentially generated by the need of the new visual tools still having to generate classical code to be used in still classical hardware platforms.

II. STATE-OF-THE-ART

A. Development software tools

Current software development tools are proprietarily built or made out of commercial tools, inside car and car-components manufacturers. While user-interfaces are now more visual and better suited for the data-flow nature of automotive functionalities, highly complex layers are still hidden under the hoods of those tools. Historically evolved through adding new visual interface layers on top of the already existing classical layer stack, these huge, very complex, and expensive software packages, are illustrated through these state-of-the-art examples:

1) *ASCET*: short for "Advanced Simulation and Control Engineering Tool", developed at ETAS [11] since 1997. This tool displays a graphical interface (Fig. 2) which allows to visually edit/design automotive functionalities called FDEFs (Function-DEFinitions). Interestingly enough, it combines representations of both data- and control-flows into a single visual language based on graphical processing elements, while having the capacity of displaying both flows simultaneously on the same screen view [13]. This tool is used for series-cars prototyping and development, while also being applied to Motorsport [14] projects.

2) *MATLAB Simulink:* short for "MATrix LABoratory" and "Simulation and Link", developed at Mathworks [7] since 1984. Very similar to *ASCET* in terms of the graphical interface and internal mechanisms, it also allows designing automotive functionalities through the special data-flow tool-box *Simulink* (Fig. 3). Although not specifically designed for the automotive scene as the *ASCET* tool is, it is without doubt the most used and mentioned tool package in the automotive and non-automotive industries, papers and general research. This tool is being applied on most recent motorsport ECUs as in [15] [16] with a combination of micro-controller plus FPGA-based hardware.

3) *LabVIEW:* short for "Laboratory Virtual Instrumentation Engineering Workbench", developed at National Instruments [8] since 1986. With evident similarities to *MATLAB Simulink* and *ASCET* (Fig. 4), it is mainly used for industrial and laboratorial applications, data acquisition and instruments control. It also allows processing on both micro-controller and FPGA-based hardware platforms.

Integrated auto-code generators produce classical "C" code usable on classical hardware, with the particularity of *Matlab* and *LabVIEW* being able to also target FPGA platforms. These additionally "sandwiched" auto-code generators [12] build the necessary interface between the graphical elements and the underlying "C"-code specialized classical tool-chains (left of Fig. 5). Corresponding centralized hardware platforms (right of Fig. 5) close this type of packages. This tool combination leads to most of the limitations and pitfalls in these highly complex systems.

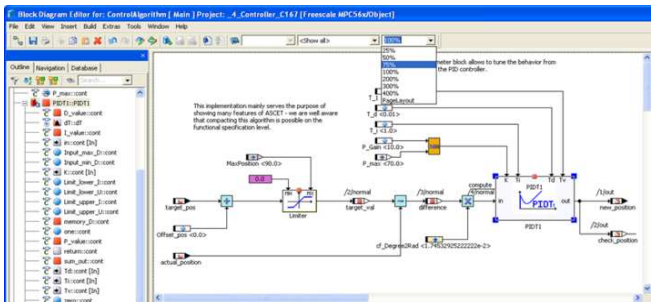

Fig. 2 - ASCET automotive-specific visual development tool.

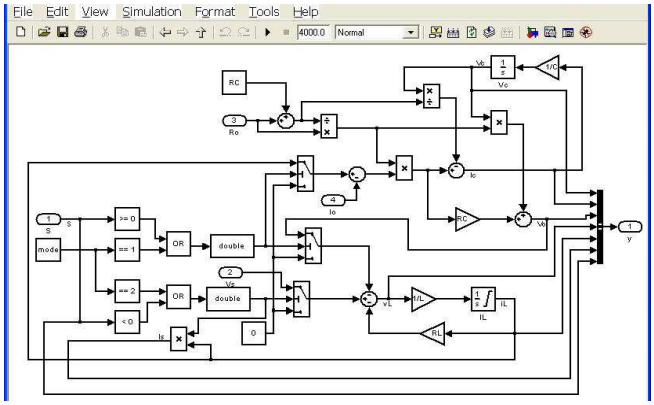

Fig. 3 - MATLAB Simulink general-purpose visual development tool.

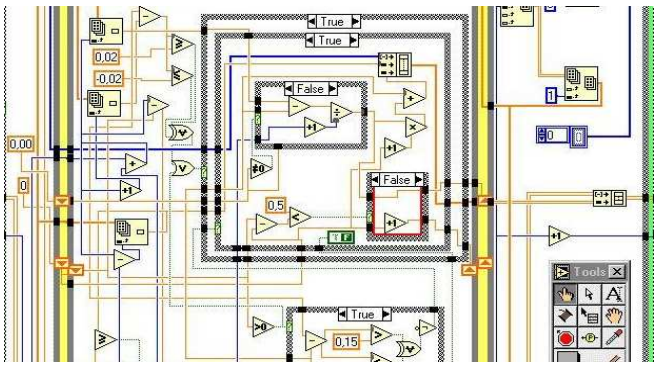

Fig. 4 - LabVIEW's general-purpose visual development tool.

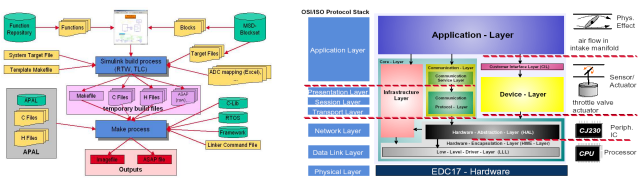

Fig. 5 - Matlab tool-chain and series-car EDC17 ECU internal structure.

In contrast to legacy but still used "C"/Assembly-only tools (Fig. 6) the visual counterparts are often referred to as "Rapid" or "Fast-Prototyping" tools, mainly because of the relative ease and speed those visual programming methods allow for users to make changes. Changes are simple visual manipulations, while underlying auto-code generators do the heavy work automatically. Delays between changes and hardware response depend on the extent of the changes and typically range from half a minute to a few minutes [15]. These tools are thus limited by minimum operating turn-around delays, imposed by their internal structure.

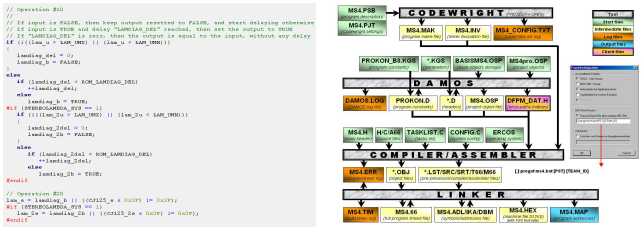

Fig. 6 - Example of a legacy ECU development software tool-chain.

Although iconic visual handling has to be learned [32] [33], its commercial success states its usefulness from the users' point-of-view, for programming complex equipment including automotive ECUs. The ability of visually conveying intuitive information is an advantage. It is also being applied to robotic equipment, with self-explaining and highly intuitive icons and programming environments [34] [35], also including script-like programming abilities.

B. Hardware target platforms

Besides proprietary hardware made directly inside the car or car-components manufacturers, several commercial platforms are also available. Special micro-controllers contain dedicated I/O peripherals, besides other mechanisms to accelerate basic routines for injection- and ignition-outputting. FPGAs are also starting to invade special highly flexible prototyping hardware. Some currently used hardware platforms are the following:

1) *Micro-Controllers*: special micro-controller families such as the C167 and the TriCore were developed by Infineon-Siemens [17] for automotive ECUs. Other automotive micro-controllers are the MC and MPC families from Motorola [18]. Although hard to program, debug and maintain, the vast majority of series-car ECUs use these "turbo-charged" feature-packed micro-controllers as their processing units. Their commercial success has mainly to do with mass-production cost-optimization, by selling the same costly and complex software millions-fold.

2) *FPGAs:* motorsport ECUs and specialized prototyping hardware tend to use concepts which allow developers to more quickly and efficiently respond to customer requests. Thus, platforms have been advancing into "programmable hardware" since the early 2000's, with Magneti-Marelli's "FastPRO", Bosch's "MS5" and dSpace's "RapidPro". These combine MATLAB Simulink, auto-code generation and standard "C" [6] [14] [16], exception made to National Instruments' FPGAs modules using LabVIEW [9].

Although FPGAs have made hardware more flexible, problems with overwhelming software tools needed to program/configure them continues to be a major pitfall in the industry. Usage of FPGAs does not get hardware much closer to the data-flow visual nature of user-interfaces, since it is still configured with VHDL or similar languages. This leaves the conceptual mismatch between higher and lower layers as an issue to still be solved or enhanced.

Despite previous state-of-the-art development software tools and hardware target platforms displaying high degrees of flexibility, still much is desirable to achieve. We herein define a rarely mentioned but very useful situation in software development: *"Trial* (not necessarily 'and error') *Reprogramming"*. This concept bases on finding the best code by repeatedly/recursively tuning the same small block of code until best results are achieved. This is done by allowing the programmer to manipulate the system with very low or virtual no turn-around delays. This kind of *Live-Programming* would allow the programmer to keep tuning or trying on the same block of code, without ever leaving it or having to restart anything in the program. Microsoft's Visual Basic/C# and in some way Sun's Java Virtual Machines referred to in the next section, nearly allow this, but still in a limited fashion.

III. COMMERCIAL & RESEARCH EFFORTS

The industry naturally tends to use well-established systems, either directly or through "evolutionary" changes, even though technically not being suited for their applications. An example is the use of *wifi* packages for "on-the-flight" data-transfer in motorsport races. Cross-layer [51] "evolutive" adaptations cannot eliminate key-limitations completely, while even keeping overall complexity and grasping difficulties. Component-based software development [53] addresses budgets and deadlines using pre-fabricated components, but hides existing inconveniences and does not simplify the core development efforts. Complex components still produce complex systems, while automotive software is too specialized and hardware-dependent to make efficient use of this development distribution paradigm.

In response to over-complexity issues directly related to the previously mentioned software tools and hardware platforms, some interesting progress in partial solutions has been made in the recent past. These efforts in bringing true easy-handling solutions to the marketplace, present themselves mostly as short-cuts for frequent programming tasks and for efficient debugging. Currently most

interesting and promising innovations include:

Visual C#: programming language developed by Microsoft since 2001

1) [27], with features based on advanced techniques of also still used *Visual-Basic* [28]. It introduces a widely employed commercial solution for making "almost on-the-fly" code-changes. Changes are quickly recompiled and applied to the existing program. This so-called "background compiler" [29] works during editing, allowing syntax errors to be continuously highlighted and corrected, thus enabling innovative *Interactive Programming* user experiences. Because object-code is kept synchronized with the source-code most of the time, much shorter turnaround delays are possible. Although large changes demand lengthier compilations, it is a big step forward in the quest for easy-handling programming paradigms.
2) *JAVA:* programming language developed by Sun Microsystems since 1995 [30], introduces a commercial solution that shifts compilation details and its complexity into an intermediate layer or the hardware itself. This simplifies the top-most interface layer which only has needs to produce well-defined byte-codes. An "automatic runtime compiler" also called "just-in-time (JIT) compiler" then compiles and runs these JAVA byte-codes natively on a JVM (JAVA Virtual Machine) as shown in Fig. 7 or even directly on the hardware [51]. Again, short turn-around delays and fast code-changes through dynamic loading of classes [31] are also granted with these systems. The more recent C# language works in a similar way, by means of its "background compiler" and the CIL (Common Intermediate Language). Extreme extensions of the already long *JAVA* multi-layered development systems may be found in "language-to-Java" translators and compilers, which convert other languages such as C, C++, Ada, Cobol, to *JAVA* byte-code, sometimes even converting first to another intermediate language and only then to native *JAVA* byte-code [41] [42] [43].
3) *IEC 61131 PLC Standard* - this software standard unites 5 ways of representing PLC programs, in both visual and textual forms [37]. The *CoDeSys* tool [38] is a good example of a commercial application of this standard. Fig. 8 shows an example. It allows the user to program in his most familiar mode(s). Classical compiler-based, this tool has to cope with all 5 modes, producing high development/maintenance complexity.
4) *ETAS ETK & HiTEX in-circuit dProbe:* hardware emulator add-ons developed by ETAS [10] and HiTEX [39], respectively. These highly disruptive and physically intrusive solutions intend to circumvent the complete complex software layer structure, by directly manipulating hardware at micro-controller and/or memory levels. Mimicking the original hardware, they are also called "in-circuit" emulators or "by-passers" (Fig. 9). Proprietary software user-interfaces then allow for direct code and memory manipulation. Although very useful and necessary for effective debugging, it does not eliminate the need of toggling between development and debugging environments. Turn-around delays, sluggish learning-curves and high complexity are still fully present limitations.

Huge efforts are made to standardize existing software and hardware structures, attempting to reach a common standard among car- and tool-manufacturers. The strongest attempt is the AUTOSAR consortium (AUTomotive Open System ARchitecture) [36], uniting core manufacturers such as BMW, Bosch, Daimler, PSA, Ford, VW, GM/Opel and Toyota. Based on similar principles as in [53], this standard merges yet more layers into existing tool-chains such as those of ASCET and Matlab Simulink. Existing easy-handling problems remain basically untouched, reaching almost out-of-control complexity levels.

Similarly, attempts and combinations [23] [24] to have a unique development language have been made by using UML [22] and its derivative SysML [26]. This tries to produce a front-end or "tying-language" on top of existing visual and non-visual tools. Associated translation layers create ever-growing chains. Other attempts even create meta-layers with multiple internal abstractions and concretization layers [25], creating ultra-complex and multi-dimensional software architectures (Fig. 10).

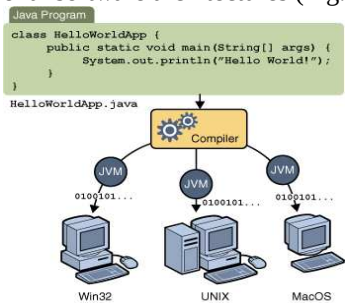

Fig. 7 - JAVA program and its byte-code execution on JVM.

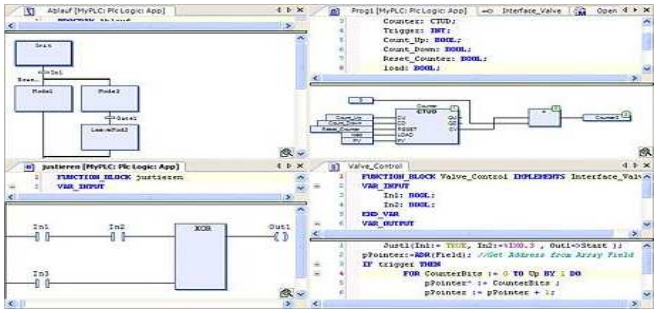

Fig. 8 - *CoDeSys* editor with multi-modal programming possibility.

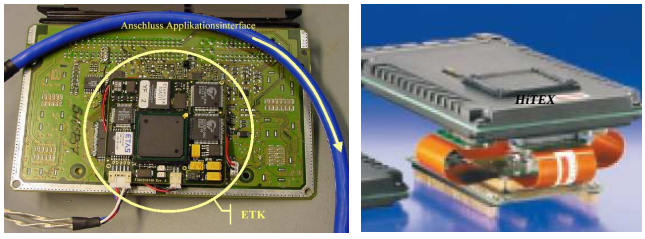

Fig. 9 - ETK and *dProbe* hardware "in-circuit" emulators

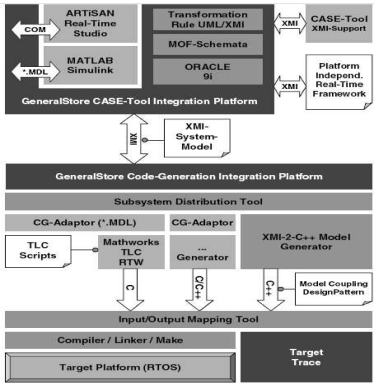

Fig. 10 - Very complex to maintain/understand multi-dimensional tool

IV. HANDLING EFFICIENCY IN DEVELOPMENT

Handling Efficiency is herein defined as the real results vs. human end-to-end effort ratio when handling software and hardware packages during system development actions. This ratio encompasses intended development efforts and actions on user-interfaces, down to the hardware platform reactions. It is desirable to be as high as possible. All previously described tools present an apparently high handling efficiency ratio. Indeed, they focus on visual user-interfaces which allow execution of almost all necessary tasks quite easily (Tab. 1) without caring about underlying mechanisms. Nevertheless, it is clear from experience that unacceptably hard troubleshooting efforts in face of internal problems might appear quickly and without notice. Focus on code-generation efficiency [12] and tool integration [19] is not enough. Despite ease of use being an actual concern [21], all current tools have very complex internal architectures demanding special expertise to troubleshoot related problems. Perpetuation of this development path reveals the infeasibility of implementing some really innovative handling methods, because it would require prohibitive overhauls of existing structures. In other words, development easiness or difficulty, associated to handling efficiency itself, is a two-fold problem.

DESIGNING	- creating, customizing and changing FDEFs
MONITORING	- viewing variable values on the FDEFs
DEBUGGING	- testing, inspecting and correcting FDEFs
PROTOTYPING	- dynamically testing and changing FDEFs
COMPARING	- viewing differences between similar FDEFs
SIMULATING	- running FDEFs on the editor w/o hardware
DEPLOYING	- applying and running FDEFs on the hardware

Tab. 1 - Most usual development tasks carried out in automotive scenes

Current software used in the automotive scene relies on classical layered approaches with user-friendly graphical interfaces on top of a long tool-chain (Fig. 11). Auto-code generators, compilers, scanners and parsers represent the "pitfall sources" in such structures. Let us also not forget the huge certification efforts [20] of auto-code generators alone. The blue line represents the desired path to achieve monitoring and debugging features (e.g. reading variables, program position) whereas the red line represents the desired path to achieve active prototyping features (e.g. changing code, algorithms, data). These paths are intrinsically difficult to establish, since they have to traverse so many different and "opaque" layers.

Starting a completely new system concept from scratch would be too expensive and risky for the automotive industry. Therefore, these historically grown "add-on-top" layered approaches prevail and are a good reason for current automotive tools being so complex and almost impossible to maintain without glitches. This is also a good reason why these tools lack ultra-fast handling mechanisms that would be highly desirable in the automotive scene. Current systems are extremely difficult to maintain and understand, turning out to be clear that these systems and other tentative approaches pitfall on internal development and also handling efficiency issues.

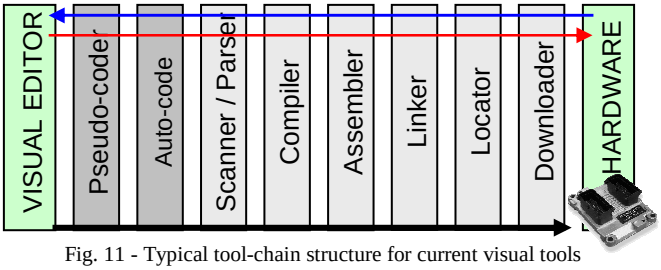

Fig. 11 - Typical tool-chain structure for current visual tools

When really advanced handling features appear in current modern tools, it is the result of huge, almost prohibitive amounts of effort, especially on the software side. Thus, historically speaking, debugging and other manipulation features tended to grow as independent assistive entities (Fig. 12), avoiding complex layer stems. Therefore, unfortunately, they are not fully integrated. These conceptually disruptive but feasible solutions rely on "layer-avoidance" approaches, rather than on the even more disruptive and complex "cross-layer" design (as unavoidably done in wireless networking [52]).

Although advances have been made on "easy-handling", our advanced concept of *"Live-Prototyping"* is still very far from reality in both automotive and commercial scenarios. Simply put, this innovative concept bases on the possibility of virtually eliminating any delays between code-changes and hardware reactions. A *100% Interactive-Programming* tool would be an obvious result. This feature would then allow concepts such as *"Trial Reprogramming"* to arise, adding a huge handling efficiency leap to existing systems.

Techniques followed up by the industry until now show that everything has been achieved through "evolutionary" approaches. These have already shown to have reached levels of highly undesired complexity. "Evolution" generally bases on developing abstraction layers over the already existing layer stack, mainly because it is infeasible to change that legacy stack. "Soft" exceptions to this rule are newer systems like JVMs, which were developed with a new view of the underlying tool-chain, but nevertheless still obeying to classical views and usage of very complex

layering, compilers and such. Thus, considering the nature of current state-of-the-art software/hardware combinations, a "clean-slate approach" seems to be the only reasonable and feasible way to redefine innovative easy-handling efficiency levels. This radical approach allows us to completely detach from all previous implementations, producing a system envisioned from scratch.

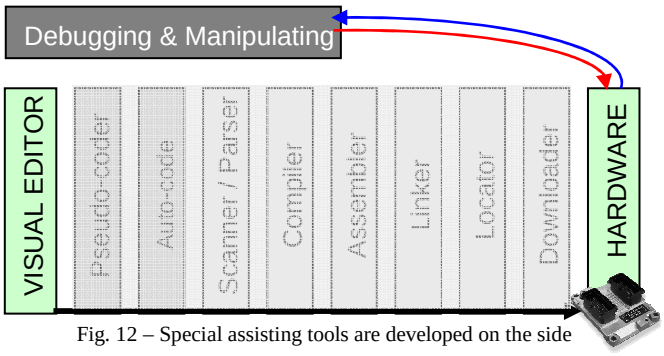

Fig. 12 – Special assisting tools are developed on the side

V. AN ADVANCED EASY-HANDLING VIEW

It is currently impossible to achieve a really disruptive *"Live-Prototyping"* programming mechanism, by simply using existing software/hardware combinations. Therefore, a new idea must be pursued to accomplish this highly innovative breakthrough. This system-manipulation concept literally means "changing an alive and running program", without ever having to stop or pause it in any way. It resembles existing *"Fast-"* or *"Rapid-Prototyping"* systems regarding simplicity of usage, but with virtually zero turnaround or implementation delays. It is not just meant that it allows online data parameter changing on a running system as in [6] [8] [11] [15] [16] (as all modern systems readily allow), but that it also allows to change the code/algorithms themselves without the running system ever noticing it or having to employ "out-of-the-box" procedures. To be truly useful, this mechanism should be 100% seamless, 100% smooth and 100% imperceptible. The hardware limits itself in working and reacting to changes, as if they were there since it ever started.

This *"Live-Prototyping"* concept compares, by its similarity, to an *Awake Craniotomy* [40], where a conscious patient's brain is being manipulated by a neurosurgeon. The neurosurgeon needs immediate conscious feedback upon all manipulations, to be able to monitor and assess the patient's neurological status during that type of surgery. It all happens without harming, cooling or in any way stopping any patient's organs (reviving them again later). All organs of the patient just keep working normally without noticing the drastic but literally life-keeping intervention.

To really accomplish the previous idea, a much more transparent and narrow software structure is needed, to allow user-interface changes to propagate rapidly and efficiently down to the hardware. This should desirably go forth without side-effects of intrusive/destructive effects.

The first step toward actually implementing this mechanism unavoidably consists of getting both ends (user-interface and hardware) as near as possible to each other. Furthermore, let us assume that it is at all possible to develop a new kind of programming language which does not need any of the complex syntax parsing, compiling, assembling and linking/locating layers. This language would thus be intrinsically both hardware- and user-interface-near. By just keeping strictly needed layers, we would reach a simplified structure depicted in Fig. 13.

A much more direct and hassle-free user↔hardware communications path would be much more suitable to allow highly useful feature, such as *Live-Prototyping*. This is again represented as red and blue arrows in Fig. 13.

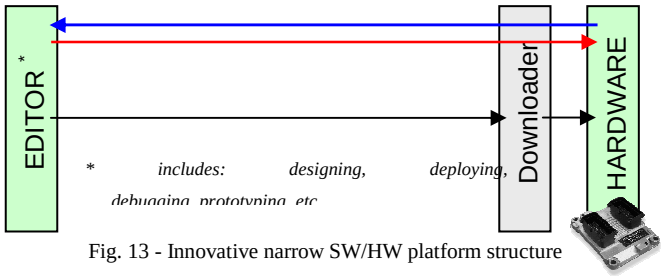

Fig. 13 - Innovative narrow SW/HW platform structure

Our intentions follow three principles: marketplace research shows that true easy-handling features are always desired by both manufacturers and customers; it turns out to be clear that attempts to derive new systems based on existing ones are impractical or impossible; automotive functionalities possess various particular traits such as data-flow orientation and mainly feed-forward processing. These principles lead to the concept presented herein.

The arising concept is the *Macro* entity. It consists of a 100% encapsulated functional/mathematical operation. It represents any of the needed automotive FDEF-operations, such as arithmetic, logic, timing, memory and conditional operations. It is immune to "soft" influences as those found in classical mechanisms containing flags, stacks, pipelines, caches, interrupts, etc. Fig. 14 illustrates such an entity, which processes values on the input *Nodes* in a fully autonomous and influence-free way, outputting the result on an output *Node*. The strict characteristics and behaviors of this fundamental *Macro* building-block follow the idea of not allowing the user to have any options or alternatives:

1) *Strict control:* the editor strictly controls, maintains and continuously enforces formal correctness of all visual elements displayed and associated to *Macros*. It also enforces automation of positioning and connecting procedures. This visual level allows no user-errors. ⇒ *Pseudo-Coder* layer turns obsolete because the *Macros* are directly used as program representation, without any pre- or lexical/syntactical processing.

2) *Strict adaptation:* all graphical *Macro* elements are themselves perfectly adapted to the subsequent layers. ⇒ *Auto-Code Generator* layer turns obsolete because *Macros* are directly used by underlying layers, without any need for complex translations or conversions.

3) *Strict native hardware:* used hardware is custom-made and fully dedicated to natively processing *Macros*. ⇒ *Scanner/Parser+Compiler+Assembler* layers turn obsolete because *Macros* are directly understood in hardware, without translation to native low-level code.

4) *Strict uniqueness:* a *Macro* just takes input values from memory, processes them and stores the result back into memory. All values are identified by unique IDs. *Macros* are executed call-less in a continuous stream. ⇒ *Linker* layer turn obsolete because there is no need to find variable addresses and prepare function-calls.

5) *Strict sequential memory: Macros*/*Nodes* are kept with sequential/unique IDs in FLASH/RAM, respectively. IDs are automatically created and applied by the editor at all time during user visual editing/manipulation. ⇒ *Locator* layer turns obsolete because *Macros* and *Nodes* are simply sequentially written into memory.

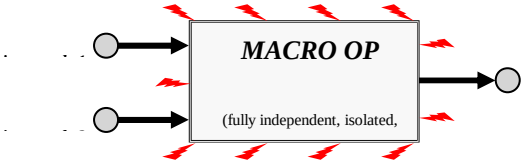

Fig. 14 - Macro entity with Nodes and immunity to external "noise"

It must be pointed out that the goal is not to develop a new general-purpose language, but one that strictly adapts to the needs of the automotive scene. This will then lead to systems which are less complex and more transparent.

It must also be emphasized that the *Macros* are strictly low-level self-contained in the sense that each *Macro* is 100% self-sufficient. Thus, they do not require any of the classical low-level entities related to flags, signals, stacks, registers, caches, pipelines and context in general, for getting the inputs successfully and correctly processed to the output. Also, the *Macros'* immunity to "soft influences" refers to the previous entities and does not include any (normally fatal) hardware failures and such.

From all previous considerations it is clear that these *Macros* are the connecting idea between the user-interface and the hardware. Thus, the missing link is the *Macros-Sequence*, similar in nature with "byte-code" in Java [30] [31]. At this precise point, it is very important to understand that everything else connects around/to this central role-playing *Macros-Sequence* depicted in Fig. 15.

At one end, the user-visible part, the visual editor, takes the unprocessed raw *Macros-Sequence* and represents it graphically through its graphics-engine. This visual format is similar to [46], using graphical elements/icons similar to the ones used in [7] [8] [11]. The editor is able to transfer user graphical manipulations to the *Macro-Sequence*, thus comprising a continuous two-way relationship. Contrary to any graphical tool (like embedded raw source-code [38]), this editor relies only on the *Macros-Sequence*, without any separately saved graphical or non-graphical information.

Finally, it is evident that the hardware platform will have to understand the *Macros* language, since no translation is made at that point whatsoever. Thus, at the other end of the system, a hard-coded *Macro-Processor* implemented in an FPGA seems to be the best choice to address highest flexibility and speed right from the beginning (as JVM [44]). This hardware thus also comprises a fast two-way relationship with the *Macros-Sequence*.

Remember that this disruptive mechanism was allowed by the adopted "clean-slate approach". In contrast, conceptual directions previously presented as state-of-the-art tools grew in a "bottom-up" (ASCET, Matlab, LabVIEW, UML) or even "top-down" (JVM, Visual C#, hardware emulators) "evolutionary" styles and methodologies.

As a final note to these ideas, *Interactive Programming* or *Live-Prototyping*, as we call it, is accomplished by those previous two two-way relationships. The *Macros-Sequence* are the privileged conceptual/central gateway to this process, keeping everything concentrated upon a single transparent and fully encapsulated entity, the *Macro* itself.

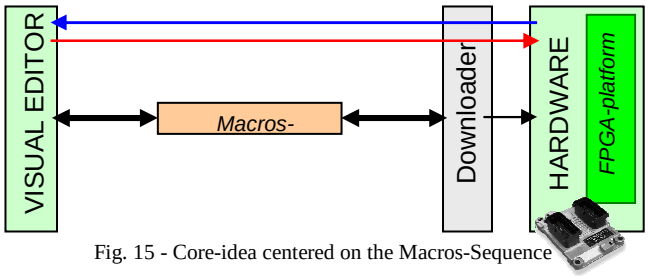

Fig. 15 - Core-idea centered on the Macros-Sequence

PRELIMINARY EFFORTS

To preliminarily test the previously exposed ideas around the *Macro*, a simple editor was programmed in Borland C++ for Windows [54] in about only 10 hours. Inside this editor the *Macro-Processor* was simply simulated in software. The editor's graphical engine was able to visually represent simple but real automotive motorsport FDEFs from their corresponding *Macro-Sequences*.

Some experimental code to test the predicted possibilities of monitoring, debugging, reverse-engineering and even simulating FDEFs, has also been implemented. Everything was seamlessly integrated and worked on the same visual user-interface depicted in Fig. 16. These preliminary tests used hand-coded *Macro-Sequences* (Fig. 16 - top-right list). The essential features available after this relatively little programming effort were the following:

- *DESIGNING* - creating, customizing, changing FDEFs
- *MONITORING* - viewing variable values on FDEFs
- *DEBUGGING* – test, inspecting & correcting FDEFs
- *STEPPING* - controlled step-wise execution of FDEFs
- *COMPARING* - viewing differences between FDEFs
- *SIMULATING* - running FDEFs online w/o hardware

Further experiments show that the followed *Macro* idea may lead to an even more "easy" way to get ultra-advanced capabilities, such as *Live-Prototyping*. This system attains such an intrinsic "transparency" by eliminating any notions

of: explicit syntax, parsing, auto-generation, compilation or assembling; static, dynamic or just-in-time compilation; virtual machines and all other forms of implicit conversion of one language formats/levels to another. All classical layers are eliminated, not just merged or transformed in any form, leaving just the indispensable user-interface and the lower download mechanisms and, finally, the hardware.

Internal details of this system have strong conceptual and even technical similarities with C#

[27], Visual-Basic [28], Java Virtual Machines [31] and auto-graphics generation directly from source code [46]. But this "center-out spreading" concept of the *"Macros-Sequence"* tries to set a new perspective on targeting development/processing systems, by allowing "true direct processing" and full "bi-directional" manipulation. All features are enabled by a very lean system without the need for "out-of-the-box" procedures or any other costly and complex mechanisms.

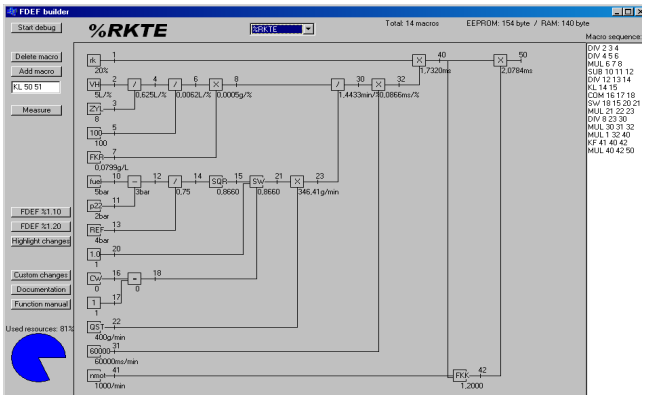

Fig. 16 - Experimental editor using a Macros-Sequence for fuel-injection

6. CONCLUSIONS

Current tools do their jobs well, but most of the time in a highly inefficient and complex manner. They use high-level syntactic or graphical languages combined with auto-code generators. These systems have historically grown through a "layered" and "evolutionary" approach, where new layers are added to accomplish more complex functions and to achieve abstracter user-interfaces. Reducing these systems to the strict user needs and to feature *Live-Prototyping*, would result in an infeasible and impractical task. This thus called for a "clean-slate approach" in an attempt of not being bogged down by "unnecessary" classical structures.

The first step was to shift the core-mechanism to the center of the system, by concentrating everything onto the *Macros-Sequence*. All other components crystallize around it (Fig. 15). Note that these appear side-ways rather than "above" and/or "under", producing a system where neither component lies above/under the others. Therefore, the conceptual *Macros* heritage spreads on the same horizontal plane. This would then be called a "center-to-the-sides" approach as compared to existing approaches which all cast into the "top-down" or "bottom-up" categories.

In this work-in-progress, the most interesting effects of having the key code-entity *Macros-Sequence* at the center of the system, feeding both user and hardware ends, go down as follows: strong "WYSIWYG" (What You See is What You Get) effect, since there are no hidden layers or features; strong "WYSIWIS" (What You See Is What I See) [32] effect, since both users and hardware see exactly the same thing (contrary to complex compilation and decompilation [45] found in other attempts).

It is not intended to develop a tool to do new things, but to do them more efficiently. Fig. 17 shows a simplified timeline of our subjective measure of *Handling Efficiency (HE)* progression over the decades for some milestone paradigms. The *HE* component purely from the users' perspective show in green and the *HE* component of the internal development efforts of the tools themselves shows in red. The largest steps upward correspond to the largest user abstraction levels, while the largest steps downwards correspond to the largest internal complexity increases. It can be seen that something has to be done in respect to the systems' internal *HE*. The *Macros* might just be a good perspective for the automotive-specific environment.

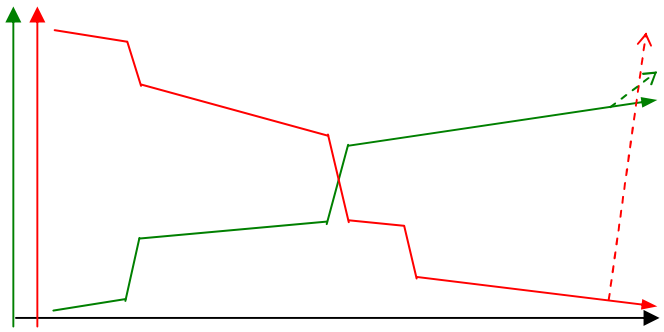

Fig. 17 - Timeline with users' (green) and internal (red) HE

7. FUTURE WORK

Live-Prototyping features will emerge in future work. These will not compare to pausing and recompiling as in C#

[27] [29] or VB [28], in recompiling at runtime [30] [31], nor to state-of-the-art *Rapid-/Fast-Prototyping* [6] [15] [16]. Distinctly from all state-of-the-art mechanisms, the *Macro-Processor* itself will not be aware of any manipulation, due to its independence from external events. This allows easy code changes on a fully alive and running system, in a fully online and seamless fashion. The conceptual complexity of the underlying supporting software and hardware structures is relatively minimal.

From the user's perspective, *Live-Prototyping* will be a 100% online and 100% seamless/transparent mechanism with zero waits or turnaround delays. This feature will naturally respect existing real-time constraints and issues at hardware level. For the first time, it will be possible for the user to accomplish all possible programming, prototyping, simulation, debugging and verification actions, in a truly

alive system, just like in an *Awake Craniotomy* [40].

In the case of an automotive system, this feature means literally the possibility of changing algorithms, general code and data, while the engine is normally running. This poses safety issues at the level of potential engine damage, of course. But these are not much different from the issues that currently arise from changing data parameters while the engine is being tuned on a dynamometer, especially concerning ignition parameters. The handling advantage of having immediate engine feedback upon data-changes is highly desirable and recognized. Live algorithm changes just adds a new dimension to this kind of easy-handling.

Since state-of-the-art automotive systems are highly distributed, with dozens of ECUs all around, connected through communications networks, it must be emphasized that *Live-Prototyping* will first center on the centralized ECU itself. While not addressing similar live manipulation on distributed systems, this possibility will eventually be contemplated later on. Clearly, communication and timing issues will be of maximum importance there.

Future work will also contemplate a "hard-wired" *Macro-Interpreter* in FPGA-based hardware. At this point, no other translating or processing interface will exist between the *Macros-Sequence* and physical hardware. Attempts of creating this kind of micro-coding direct processing hardware have already been made [47] [48] [49]. This dedicated hardware goes entirely in the direction of a 40-year-old interesting philosophical paper [1], by finally building hardware designed to directly accomplish the tasks it will really be assigned to.

It is expected that with these previous considerations, a system may be built, comprising the live operation of a small gasoline combustion engine. This encompasses a complete visual editor user-interface suite integrating all desired actions, as well as a complete hardware prototype able to drive all necessary sensors and actuators of the engine. All this will be merged with the dual feature of *"Live-Prototyping"* / *"Trial Reprogramming"*.

REFERENCES

[1] W. McKeeman, "Language directed computer design", AFIPS Joint Computer Conferences Proceed., November 14-16, 1967, p413-417.

[2] J. Friedman, "MATLAB/Simulink for Automotive Systems Design", Design, Automation and Test in Europe, 2006. DATE '06. Proceed.
Volume 1, 6-10 March 2006 Page(s):1 - 2.

[3] S. Sims, R. Cleaveland, K. Butts, S. Ranville, "Automated validation of software models", Automated Software Engineering, ASE2001 Proceed. 16. Annual Int'l Conference 26-29 Nov2001, 91-96.

[4] M. Kuhl, C. Reichmann, I. Protel, K. Muller-Glaser, "From object oriented modeling to code generation for rapid prototyping of embedded electronic systems", Rapid System Prototyping, 2002 Proceedings, 13th IEEE Int'l Workshop on 1-3 July 2002, p108-114.

[5] dSpace GmbH, www.dspaceinc.com.

[6] dSpace, "dSpace RapidPro: Full Power", www.dspaceinc.com.

[7] The MathWorks Inc., "MATLAB Simulink", www.mathworks.com.

[8] National Instruments, "LabVIEW", www.ni.com.

[9] National Instruments, "Neue Funktionen für LabVIEW, parallel, drahtlos und in Echtzeit", 2009, www.ni.com.

[10] ETAS GmbH, "ETK/XETK ECU Interfaces", www.etas.com.

[11] ETAS GmbH, "ASCET Software Family", www.etas.com.

[12] U. Honekamp, J. Reidel, K. Werther, T. Zurawka, T. Beck, "Component-node-network: three levels of optimized code generation with ASCET-SD", Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on 22-27 Aug. 1999 Page(s): 243-248.

[13] H. Randriamparany, B. Ibrahim, "Seamless integration of control flow and data flow in a visual language", Computer Systems and Applications, ACS/IEEE International Conference on. 2001, 25-29 June 2001 Page(s): 428-434.

[14] Bosch Motorsport, www.bosch-motorsport.com.

[15] Bosch Motorsport, "MS5 ECU family", www.bosch-motorsport.com.

[16] Magneti-Marelli, "FastPRO ECU family", www.magnetimarelli.com.

[17] Infineon-Siemens, "C167/TriCore µC families", www.infineon.com.

[18] Motorola, "MC and MPC µC families", www.motorola.com.

[19] M.H. Smith, M. Elbs, "Towards a more efficient approach to automotive embedded control system development", Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on 22-27 Aug. 1999 Page(s):219 - 224.

[20] T. Glötzner, "IEC 61508 Certification of a Code Generator", System Safety, 2008 3rd IET Int'l Conference on 20-22 Oct2008, 1-4.

[21] I. Bell, "Software - More than a programming language - Whether testing cars at automotive companies or controlling production and quality at manufacturing plants, engineers and scientists need flexible, cost-effective solutions for test, measurement and automation", Computing & Control Engineering Journal Volume 18, Issue 1, Feb.-March 2007 Page(s): 26 - 29.

[22] Object Management Group (OMG), "Unified Modeling Language (UML)", www.uml.org.

[23] L.B. Brisolara, M.F.S. Oliveira, R. Redin, L.C. Lamb, F. Wagner, "Using UML as Front-end for Heterogeneous Software Code Generation Strategies", Design, Automation and Test in Europe, 2008. DATE '08, 10-14 March 2008 Page(s): 504 - 509.

[24] T. Farkas, E. Meiseki, C. Neumann, K. Okano, A. Hinnerichs, S. Kamiya, "Integration of UML with Simulink into embedded software engineering", ICCAS-SICE 2009, 18-21 Aug.2009 Page(s): 474-479.

[25] C. Reichmann, M. Kiihl, P. Graf, K.D. Muller-Glaser, "GeneralStore - a CASE-tool integration platform enabling model level coupling of heterogeneous designs for embedded electronic systems", Engineering of Computer-Based Systems, 2004 Proceedings. 11th IEEE Int'l Conf. and Workshop on 24-27 May 2004 pp: 225-232.

[26] Object Management Group (OMG), "Systems Modeling Language (SyML)", www.omgsysml.org.

[27] Microsoft, "C# Language Specification 3.0", www.microsoft.com.
[28] Microsoft "Visual-Basic Language Specs", www.microsoft.com.
[29] M. Gertz, "Scaling Up: The Very Busy Background Compiler", MSDN Magazine, Microsoft, msdn.microsoft.com.
[30] T. Lindholm, F. Yellin, "The JavaTM Virtual Machine Technology Specification", 2.ed., Sun Microsystems, www.sun.com.
[31] Sun Microsystems, "Java HotSpotTM Virtual Machine", sun.com.
[32] M. Petre, "Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming", Comm. of the ACM, Jun.1995, pp. 33-44.
[33] A. Ko, B. Myers, A. Htet, "Six Learning Barriers in End-User Programming Systems", Visual Languages and Human Centric Computing, 2004 IEEE Symposium 30-30 Sep.2004 pp:199-206.
[34] R. Bischoff, A. Kazi, M. Seyfarth, "The MORPHA style guide for icon-based programming", Robot and Human Interactive Communication, 2002. Proceedings. 11th IEEE International Workshop on 25-27 Sept. 2002 Page(s):482 - 487.
[35] MORPHA Konsortium - Kommunikation, Interaktion und Kooperation zwischen Menschen und intelligenten anthropomorphen Assistenzsystemen, www.morpha.de.
[36] AUTOSAR-Automotive Open System Architecture, autosar.org.
[37] International Electrotechnical Comission, "IEC 61131-3 standard for PLC programming languages", www.iec.ch.
[38] Smart Software Solutions, CoDeSys, www.3s-software.com.
[39] HiTEX Development Tools, "In-Circuit Emulators", www.hitex.com.
[40] V. Bonhomme, C. Franssen, "Awake craniotomy", European Journal of Anaesthesiology: Nov. 2009, Vol. 26, Issue 11, 906-912.
[41] NestedVm, Binary translation for Java, nestedvm.ibex.org.
[42] C2J Translator, C-programs to Java-programs, novosoft-us.com.
[43] S. Malabarba, P. Devanbu, A. Stearns, "MoHCA-Java: a tool for C++ to Java conversion support", Software Engineering. Proceedings of the 1999 Int'l Conference on 22-22 May 1999 pp: 650-653.
[44] S. Nino, T. Mori, YoungHun Ko, Y. Shibata, K. Oguri, "FPGA Implementation of a Statically Reconfigurable Java Environment for Embedded Systems", Field-Programmable Technology, 2007. ICFPT 2007. International Conference on 12-14 Dec2007 Page(s): 317-320.
[45] J. Miecznikowski, L. Hendren, "Decompiling Java using staged encapsulation", Reverse Engineering, 2001. Proceedings. Eighth Working Conference on 2-5 Oct. 2001 Page(s): 368 - 374.
[46] H. Prahofer, D. Hurnaus, C. Wirth, H. Mossenbock, "The Domain-Specific Language Monaco and its Visual Interactive Programming Environment", Visual Languages and Human-Centric Computing, VL/HCC. IEEE Symposium 23-27 Sep2007 pp104-110.
[47] G. Chelsey, W. Smith, "The hardware-implemented high-level machine language for SYMBOL", AFIPS Joint Computer Conferences Proceedings, May 18-20, 1971, Pages 563-573.
[48] J. Anderson, "A computer for direct execution of algorithmic languages", AFIPS Joint Computer Conferences archive Proceedings, December 12-14, 1961, Pages 184-193.
[49] H. Weber, "A microprogrammed implementation of EULER on IBM system/360 model 30", Communications of the ACM, Volume 10, Issue 9 (September 1967), Pages: 549 - 558.
[50] The Microengine Company, "WD/90 Pascal MICROENGINE Reference Manual", preliminary edition, 1979.
[51] C. Porthouse, "High performance Java on embedded devices - Jazelle DBX technology: ARM acceleration technology for the Java Platform ", Jazelle DBX WhitePaper, October 2005, ARM Limited.
[52] V. Srivastava, M. Motani, "Cross-layer design: a survey and the road ahead", Communications Magazine, IEEE Vol.43, Issue 12, Dec. 2005 Page(s): 112 - 119.
[53] P. Vitharana, "Risks and challenges of component-based software development", Aug 2003, Communications of ACM, Vol.46 Issue 8.
[54] Borland C++ for Windows, www.borland.com.