

A Vision for Embedded Software

Alberto Sangiovanni-Vincentelli
University of California at Berkeley
515 Cory Hall
Berkeley, CA. 94720
+1 510 642-4882
alberto@eecs.berkeley.edu

Grant Martin
Cadence Design Systems
555 River Oaks Parkway
San Jose CA 95134 U.S.A.
+1-408-894-2986
gmartin@cadence.com

ABSTRACT

In this paper we describe a vision for the future evolution of Embedded SW (ESW) design methodologies as part of overall Embedded Systems (ES) development. Fundamentally, we believe that the way in which embedded SW is developed today must change radically. The key steps are: first, to link embedded software upwards in the abstraction layers to system functionality; and second, to link embedded software to the programmable platforms that support it. This will provide the much-needed means to verify whether the constraints posed on Embedded Systems are met. We envisage an optimised, automated, transparent and mathematically correct flow from product specification through to implementation for SW-dominated products implemented with highly programmable platforms.

1. INTRODUCTION

Embedded Software (ESW) design is just one, albeit critical, aspect of the more general problem of Embedded System Design (ESD or just ES). ESD is about the implementation of a set of functionalities satisfying a number of constraints ranging from performance to cost, emissions, power consumption and weight. The choice of implementation architecture implies which functionality will be implemented as a hardware component or as software running on a programmable component. In recent years, the functionalities to be implemented in ES have grown in number and complexity so much so that the development time is increasingly difficult to predict and to keep in check. The complexity increase coupled with the constantly evolving specifications has forced designers to look at implementations that are intrinsically flexible, i.e., that can be changed rapidly. Since hardware-manufacturing cycles do take time and are expensive, the interest in software-based implementation has risen to previously unseen levels. The increase in computational power of processors and the corresponding decrease in size and cost have allowed moving more and more functionality to software. However, this move corresponds to increasing problems in verifying design correctness, a critical aspect of ESD since several application domains, such as transportation and environment monitoring, are characterised by safety considerations that are

certainly not interesting for the traditional PC-like software applications. In addition to this aspect, little attention has been traditionally paid to hard constraints on reaction speed, memory footprint and power consumption of software. This is of course crucial for ES. These considerations point to the fact that ESW is really an implementation choice of a functionality that can be indifferently implemented as a hardware component and that we cannot abstract away hard characteristics of software as we have done in the traditional software domain. No wonder then that we are witnessing a crisis in the ES domain for ESW design. This crisis is not likely to be resolved going about business as usual but we need to focus at the root of the problems.

Our vision for ESW is to change radically the way in which ESW is developed today by: 1) linking ESW upwards in the abstraction layers to system functionality; 2) linking ESW to the programmable platforms that support it thus providing the much needed means to verify whether the constraints posed on ES are met.

To realise our vision, we have on one hand to develop formal techniques at the abstract level so that verification is started early and with the correct set of tools and methods. On the other hand, we have to think of ESW and hardware architecture in a unified and harmonious way. In addition to the pressure for system designers to choose flexible means of implementation, we are witnessing also a growing attention of the IC manufacturers towards chips that can be shared across several designs so that the development cost, which is also increasing by leaps and bounds as manufacturing processes evolve below the .2 micron barrier, could be amortised over a large number of units. This alignment in planets is the cause of the birth of platform-based design [1,2] where re-use and programmability are the name of the game. Programmability here will also extend to hardware implementation with the advent of embedded FPGA that will allow a system designer to use another design trade-off point where functionalities can be allocated to programmable hardware blocks in addition to hardware and software. The concept of platform is related to System-on-Chip (SoC) but it is by no means a synonym. Indeed, a platform can actually manifest itself as a chip-set, perhaps a System-in-Package (SiP), or an embedded system-board or higher-level object. *A platform is by definition something to build upon.* In fact, in most of the manifestation of a platform we notice that the actual hardware architecture is not fully specified leaving some degrees of freedom to system designers to pick components for the final implementation from a pre-set menu. Of course, the indispensable presence of programmable cores requires chipmakers to acquire a new (for them) expertise in ESW. In this case, they need to provide the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CASES 2001, November 16-17, 2001, Atlanta, Georgia.
Copyright 2001 ACM 1-58113-000-0/00/0000...\$5.00.

users of their platforms means for mapping quickly and effectively their desired functions onto the components of the platform. Today, most of the platform providers rely upon third parties to provide a tool chain for programming the cores but there are many signs that this business model needs to change: third parties are often too small and not equipped to provide supports for hundreds of users and applications and they are not necessarily interested in developing tools for new architectures. Hence, we are witnessing on one side an increasing consolidation in the industry where companies like WindRiver are growing by acquiring other Real-Time Operating System companies to capture a sizable chunk of the market that will allow them to think strategically about this very appealing and growing market. On the other side, we see semi-conductor companies acquire more competence in software tools.

We believe that realising this vision for embedded SW requires combining the strengths of several communities of interest: researchers in system level design and Electronic Design Automation (EDA); researchers in embedded SW and systems; and tool companies in both the EDA and ESW domains. These worlds have traditionally been isolated; it is time for the worlds to collide.

We have been active in researching along these lines at University of California at Berkeley under the Gigascale Silicon Research Center and at the Cadence Research Labs. In this paper, we outline the principles of our work and our research agenda.

2. TODAY'S EMBEDDED SW

ESW today is written using low level programming languages such as C or even Assembler to cope with the tight constraints on performance and cost typical of most embedded systems. The tools available for creating and debugging software are no different than the ones used for standard software, compilers, assemblers and cross-compilers. If any difference can be found, it is in their quality: most tools for ESW are rather primitive when compared to equivalent tools for richer platforms. On the other hand, ESW needs hardware support for debugging and performance evaluation that in general is not a big issue for traditional software. In most embedded software, operating systems were application dependent and developed in house. Once more performance and memory requirements forced this approach.

When embedded software was simple, there was hardly need for a more sophisticated approach. However, with the increased complexity of ES applications, this rather primitive approach has become the bottleneck and most system companies have decided to enhance their software design methodology to increase productivity and product quality. However, we do not believe that the real reasons for such a sorry state are well understood. We have seen a flurry of activities towards the adoption of object-oriented approaches and other syntactically driven methods that have certainly value in cleaning the structure and the documentation of embedded software but have barely scratched the surface in terms of quality assurance and time-to-market. Along this line, we also saw a growing interest towards standardization of Real-Time Operating Systems either *de facto*, see for example the market penetration of WindRiver in this domain, or through standard bodies such as the OSEK committee established by the German automotive industry. In fact, the single

largest market segment in the ESW arena is about RTOS, hence IP-dominated. Quite small still is the market for development tools even though we do believe this is indeed the place where much productivity gains can be had. There is an interesting development in some application area where the need to capture system specifications at higher levels of abstraction is forcing designers to resort to tools where software is clearly an implementation aspect and pose their emphasis on mathematical descriptions. The leader in this market is certainly the Matlab tool set developed by MathWorks. In this case, designers develop their concept in this friendly environment where they can assemble their designs quickly and simulate their behaviour. While this approach is definitely along the right direction, we have to raise the flag and say that the mathematical models supported by Matlab and more pertinently by Simulink, the associated simulator, are somewhat limited and not covering the full spectrum of embedded system design. The lack of data flow support is critical. The lack of integration between the FSM capture tool (State Flow) and Simulink is also a problem. As we will reason in more details later, this area is of key interest for our vision. It is at this level that we are going to have the best results in terms of functional correctness and error free refinement to implementation. The understanding of the mathematical properties of the embedded system functionality must be a major emphasis of this vision.

In the embedded system world, the increasing power of integrated circuit technology, in terms of computing power, sensor and actuator technology, mechatronics, optics and power consumption has created yet another war zone. Thus, when we address the issue of ESW design we cannot forget about the supporting technology that is probably the main reason for the need of a radically new approach.

In this situation, the boundaries between systems and IC's are blurring: what we thought to be a system yesterday that needed a PCB as supporting hardware has become a single chip solution. If this is the case, more and more burden is placed on the IC provider that has to understand system engineering and has to become himself a subsystem provider. With the economics of IC manufacturing, IC makers are turning towards programmable platforms (often dubbed as System-on-a-Chip) to increase their revenues and income. However, they cannot sell "bare silicon" to system houses as they used to. They have to provide a number of supporting elements: if they provide a platform with an embedded programmable core then they are expected to offer also the software development tools and a set of IP's that make the job of the system implementer easier. The ball game is moving from hardware manufacturing excellence to a balanced approach where hardware manufacturing, programming tools, and IP's are all important factors. Today, IC makers often resort to third parties to support their products both in terms of IP's and tools. Thus, the system designer is confronted with the nightmare of coping with different user interfaces, programming styles, IP documentation styles and contents. Thus, we can say that system designers are probably worse off than a few years ago.

In summary, the system houses have a strong need to increase software productivity and quality while facing increasing complexity in the requirements their customers place upon them and of increasing complexity in the hardware they have to use to improve performance and meet the constraints. The IC makers have the need to develop platforms that can be promptly and

effectively used by system houses in short time and with as little risk as possible in terms of market share and adequacy of their architectures. If these two sides of the electronics world do not come together and share a common view of embedded system design we believe there will be little hope of success. However, if they do, then there is a great future shining in front of us where a real co-design methodology can be envisioned. In this future, SoCs will be designed with a good characterisations of the applications that will run on it, where ESW will be designed with a clear understanding of the capability of the SoC and of the trade-offs involved in implementing a feature in hardware or software. Today, the partition of features among components of the SoC is heuristic at best. There is also a strong movement towards intermediate solutions for a new breed of SoCs where the software programmable cores and the hardware blocks are integrated with intermediate devices such as re-configurable hardware. FPGAs come to mind where flexibility is conjugated with decent performance and power consumption. Re-configurable processors are also gaining acceptance where the instruction set implemented by the processor can be customised to the implementation domain without the need of re-designing the processor itself. Processors that have fine granularity parallelism such as VLIW are also coming of age and present a threat for high-performance DSP applications. Yet how do we decide which solution is the best for our embedded system? How do we map our application onto these somewhat esoteric architectures? Our vision in this area is based on the fundamental introduction of function-architecture co-design [1,3]: how to choose an architecture for a particular system application, how to give constructive feedback to the platform provider, how to evaluate the performance of an architecture for a given application software, how to dimension a platform and to choose a scheduling mechanism so that real-time performance can be met.

3. A EUROPEAN PERSPECTIVE

We are not alone in worrying about research in ESW, DARPA in the US and the European Community are deeply concerned about the state of affairs in embedded system design. Their funding for this field is going to increase significantly. Europe in particular has always been sensitive to issues touching upon system level design given the structure of their electronic industry that is much heavier on the system side than on the semiconductor side. To quote some of the European initiatives, we looked at the "Technology Roadmap on Software-Intensive Systems: The Vision of ITEA" [4]. The major challenges for System Engineering, Software Engineering and Engineering Process Support noted by ITEA include:

- Increase Reuse
- Hardware/Software co-design
- Modelling non-functional properties
- Move from products to services via Software Components
- System and SW architecture
- Validation and verification at the system level
- Adaptation of HW and SW via re-configurable architectures and component plug and play
- Composable SW systems using reusable SW components, integration and certification of components

- Support of parallel development via integration technology
- Development of common workflow and process standards in semantics.

ITEA also looked at possible evolution trends, and highlighted:

- Formal verification, co-design flows, reuse and integration of components, (this was highlighted as a major need), platform/function co-design, unified HW/SW flows, domain/architecture co-design, and many others. In particular, one might quote "Component reuse standards to allow intellectual property design to be introduced into the embedded software design world", a quiet and subtle nod to the fact that in HW, IP design and reuse is more common than in embedded SW. Several key issues in efficient component reuse are configurability (to allow optimised reuse), decomposition and modularity, and carefully crafted interfaces.

4. REALISING THE VISION

We already reasoned about some of the concerns that are typical of ESW that are not commonly understood in the software community. In addition, we stress that the research agenda has to address these very issues if we wish to come out of the ESW quagmire. The approach we advocate is holistic: it includes methodology, supporting tools, IP's, hardware and software platforms, and supply chain management. Only by taking a global, high-level view of the problem, can we devise solutions that are going to have a real impact on the design of embedded systems. The essential issue to resolve is the link between functionality and programmable platforms. We need to start from a high-level abstraction of system functionality that is completely implementation independent and rests upon solid theoretical foundations that will allow formal analysis. We also need to select the platform that can support the functionality meeting the physical constraints placed on the final implementation. We need to implement the functionality onto the platform so that its properties of interest are maintained and the physical constraints are indeed met. In summary, our goal is:

To have an optimised, semi-automated, transparent, verifiable and mathematically correct flow from product specification through to implementation for SW-dominated products implemented with highly programmable platforms.

Highly programmable platforms can include analogue components, re-configurable logic, and specialised ASIC implementations, programmable processors of all types, configurable processors, and dedicated HW and SW blocks drawn from libraries.

Our research goal can be decomposed into smaller goals covering the major stages of design, implementation and verification:

4.1 Specification

The term "specification" means different things to different people. We understand specification to be the entry point of the design process. In our opinion, a specification should contain three basic elements:

1. A denotational description of the functionality of the system, i.e., this description does not imply an implementation. In general, the denotational description should be expressed as

a set of equations and inequalities in an appropriate algebra. Finding a satisfying set of variables corresponds to “executing the specifications”. This unambiguous interpretation of the description of the functionality allows to assess properties and to evaluate the relationships among variables of the design. Note that properties can be interpreted as abstracted behaviours of the denotational description where some of the variables and equations are abstracted away. In some sense, properties are redundant and needed to make sure that the denotational description is “correct” according to the criteria summarised by the properties. In this respect, FSM specification, data flow models such as Static (also called somewhat improperly Synchronous) Data Flow, and Event Driven models are imperative specification, where the communication mechanism and the firing rules of the elements are fully specified. In this case, we like to think of these specifications as refinements of the denotational description. It is important to devote attention to imperative specifications but with always with an eye to the relationship to the denotational specification, which, in our opinion, is a cleaner and more rigorous way of looking at system design.

2. A set of constraints that have to be satisfied by the final implementation of the system. In general, this set of constraints can be expressed also as a set of equalities and inequalities but the variables appearing in the expression are uninterpreted since they involve quantities that have no meaning in the denotational description. For example, when we specify the behaviour of an automotive engine controller we cannot refer to its weight, cost, emissions or size: these variables are a function of a physical implementation. These constraints can be seen as restricting the space of possible implementations of the functionality.
3. A set of design criteria. These are expression of design variables but are not bound. The goal of the designer is to obtain an implementation of the system whose behaviour satisfies the denotational description and the set of constraints, and optimises the criteria. Hence, the difference between constraints and criteria is that constraints have to be met, while we try to do our best to optimise the criteria. Note that when we have multiple criteria, it is often the case that they are conflicting. Hence, the optimisation problem is a very complex one.

An open question is the language used to express these elements. We believe that the most important issue is not so much the “language” *per se* but its semantics. In fact, the issue of the mathematical models and of the algebra used to compose and manipulate the models is the essential part of this stage of the design process. The choice of languages is often a compromise among several needs, for example, ease of expression, compactness and composability. We believe that the specification languages will be heterogeneous since each application domain will require different constructs even though the mathematical underpinnings are likely to be common. Today, the mathematical frameworks are partitioned in different domains, i.e., FSMs, data-flow graphs and discrete event systems, each having a particular set of intrinsic properties and execution models that determine the simulation mechanisms used to evaluate the description. While these formalisms are mathematically rigorous, there is a great deal of difficulty in composing heterogeneous specifications. The

mathematical framework that must be developed will allow understanding the relations among heterogeneous models and the composition rules and assumptions. This mathematical framework will allow us also to establish properties of the composition of heterogeneous models that are impossible today.

In this domain, there is strong interest in UML as a standard expression mechanism. At this time, while UML provides some degree of standardisation in terms of the syntactical aspects of system design, its semantics is too generic to be useful in our framework. Hence, we have two choices: define our own syntax for system level languages or adopt UML but enriching its definition with mathematical properties related to the models of computation. By focusing on the semantics aspects, we will be able to stay above the language debate and to participate to any activity evolving from standardisation efforts. We have begun to sketch the outlines of an “Embedded UML” profile that would be a basis for embedded software development [5]. Something like this is essential to bridge the gaps between the traditional object-oriented software development community and this new vision. Another interesting possibility for capturing denotational constraints is the Accellera ‘Rosetta’ language [6].

4.2 Refinement and Decomposition

Once a specification is captured, the design process should progress towards implementation via well-defined stages. The essential idea is to manipulate the description by introducing additional details while preserving the original functionality and its properties and meeting the constraints that can be evaluated at the new level of abstraction. The “smaller” the steps are, the easier it is to prove formally that constraints are met and properties satisfied. This process is called successive refinement and in our opinion should be one of the mainstays of the ESW design methodology. The issues to resolve are the precise definition of what refinement means in the appropriate mathematical framework and the proofs that the original properties are maintained. The two issues are clearly interrelated. While marching towards implementation, it is often convenient to break parts of the design description into “smaller” parts so that optimisation techniques have a better chance of producing interesting results. This process is called decomposition. The main point here is to determine whether the decomposed system satisfies the original specifications. Decomposition can be considered the inverse transformation of composition. Thus, successive refinement, decomposition and composition are the three basic operations used to move towards implementation. Both composition and decomposition are difficult when heterogeneous models are used to represent the parts to be composed or that result from the decomposition. The mathematical framework wukk provide the foundations for showing the relationship between the system before and after these transformations.

4.3 Analysis

While marching towards the final implementation, we may take paths that lead to designs that have no chance of satisfying some of the constraints. Hence, it is extremely important to have tools that evaluate intermediate results obtained during the refinement process with respect to the constraints. To do so, the essential issue is finding appropriate models at this level of abstraction that can carry enough information about the ultimate physical implementation. The alternative is going deep down in the

implementation path and use detailed models that contain the variables involved in the constraints. For example, when assessing whether a particular hardware architecture satisfies a real-time constraint, we may need to use an RTL model of the programmable processors included in the platform, in dynamic simulations. This can give accurate information but may need an inordinate amount of computing time when we just need to have a feel for the likelihood of an architectural choice of meeting the constraints. Static analysis methods such as STARS [7], and abstract dynamic models, can give us up to 1000x performance improvement, over standard simulation-based techniques on detailed models. At a high-level of abstraction, not only the simulation of the system may take orders of magnitude less computing time, but we may also be able to use formal methods to prove that performance numbers are satisfactory or not.

4.4 Target Platform Definition

Since most embedded systems are defined to map onto a target platform e.g., HW-SW architectures, and libraries, to encourage maximal reuse, it is necessary to find the right specification form and notations with which a target platform can be described - covering the full scope of its services, and configurability. This might be called the "System Platform API" conceptually. When a platform offers re-configurable logic, new methods of describing the service and configuration space are required.

One option is to develop a UML based description of a target platform that can become the target of refinement and analysis. This description needs to represent the full range of services (computation, communications, co-ordination) in both SW and HW domains offered by the platform to system implementers, including configurability options, but will emphasise an ESW view. A multi-level, hierarchical, 'stack'-oriented view of platform services can be described. Such a UML profile will deliver a view of the "System Platform API". To carry the design into the platform HW domain for more detailed analysis of system performance and more detailed HW-SW verification, it will be possible to utilise emerging C++ class-library-based languages such as SystemC (2.0 version) [8].

4.5 Mapping

The mapping paradigm associates portions of the embedded system refined specification with specific implementation vehicles contained within the target platform. Mapping is a fundamental method of configuration and refinement down to implementation. Constraint-driven mapping to a choice of implementation fabrics becomes an essential method for partitioning into HW and SW and deciding on interfaces between implementation components.

Using the System Platform API descriptions of a target platform, designs can be mapped onto the services offered by the platforms and various design space exploration analyses in the domains of performance (and possibly other domains) can be carried out.

4.6 Link to Implementation

Ideally, a platform-based design relies mostly on selection of reusable service components offered by the platform together with necessary configuration. However, derivative products often contain new or significantly modified functionality. It thus will be necessary to support SW, HW and interface synthesis to allow a comprehensive flow from specification to implementation. In addition, static configuration (selection of components and

connectivity) and dynamic control configuration (over scheduling, resource allocation and communications) are needed to control the implementation process optimally.

The implementation process involves not only the selection of reusable components chosen from a library but also the (possibly automatic) generation of software for the programmable components, of the static or dynamic configuration of reconfigurable hardware and of the synthesis of appropriate hardware modules according to performance and efficiency criteria. While we believe that the selection of the components could be done satisfactorily by hand, the rest of the work to obtain an implementation could and should indeed be done automatically to minimise human errors and to optimise productivity. The value of automatic synthesis has been proven both in software and hardware domains. In the software domain, compilers could be interpreted as synthesis tools: they map from high-level constructs into machine code. This has been the highest productivity improvement in software engineering since the advent of the digital computer. In the hardware domain, logic synthesis has had a major impact in improving time-to-market for digital designs. Most ASICs have been designed with logic synthesis tools that generate a net list of gates from a high-level language based description of the hardware to implement. In both cases optimisation has played an essential role: without optimisation, the quality of the results would have made this approach definitely inferior to a manual design process and the change in design methods almost impossible. A similar paradigm is needed here: go from a high-level representation of the functionality of the system to the detailed implementation. Without constraints, this process would be too difficult to carry out. However, since we have selected already an architecture and mapped functionality onto "virtual" blocks, the synthesis process is controlled and as such, the automatic approach conceivable.

Success here will have two major results: correct-by-construction implementations, once the transformations used in the synthesis process have been proven correct, and better quality since more attention can be spent at the higher level of the design where most of the innovations come.

The idea here will be to exploit fully the knowledge of the implementation domain. Synthesis would be very difficult, not to say impossible, if we could not leverage the work outlined in the previous sections. In particular, the design specification has to capture the mathematical properties of the behaviour we wish the system to have. The mathematical properties can be exploited to devise algorithms that can be proven correct and manipulate the description in ways that are very difficult to master by a human designer thus allowing a wider design exploration albeit constrained to the mapping suggested by the designer. Thus, one more time, we wish to stress that this vision is a holistic entity: no topic can be explored in isolation. This is the beauty and the complexity of system design.

4.6.1 SW Synthesis

Almost all the system-level design tools that comprehend software implementations (e.g., State Charts, Simulink/State Flow, ASCET, SPW) have some code generation capabilities. The approach is to start from the high-level description supported by these tools, e.g., graphical state machines, data flow graphs, flow diagrams, and to generate C code that implements the constructs of the high-level (graphical) languages. The C code can then be

used for simulation or can be output towards a C compiler for a target implementation. In some cases, the code generated by these tools is acceptable to a system designer. However, it is often the case that a quick analysis allows an experienced designer to outperform the automatic code generator. The reasons are many: the most obvious ones are the lack of a complete understanding of the abstract manipulations and optimisations that can be applied on the mathematical description and the lack of understanding of the architecture of the target programmable component. In addition, in all of these approaches, the code generation process targets single functional blocks not the complete system. There is little work on automatic synthesis of scheduling algorithms to co-ordinate the functional components so that the constraints are met. Concepts of communication/component-based [9] design give have an overall approach to cope not only with the functional blocks but also with their compositions. Some work has been done in the communication area with time-triggered architectures and with synchronous languages but much remains to be done in terms of optimal selection of communication mechanism among software components.

In the FSM domain, preliminary results that stemmed out of the work on the Berkeley POLIS system [10], and additional research, indicate that FSM-based descriptions are more amenable to automatic synthesis for two reasons. First, the mathematical model is well understood and many optimisation algorithms have been devised over the years and second, humans have difficulties to explore effectively control structures when compared to data flow like computation.

In the area of data flow graphs, some code generation work has been published over the years and results are widely available. Their quality is constantly improving but still humans can do better than automatic tools in many cases. We suspect that the problem is the global view of the structure of the computation and the understanding of the memory organization that is quite complex to take into consideration for highly optimised DSP's. This area of research is vital to improve the quality of dataflow code generation to automate SW synthesis in this domain.

4.6.2 HW Synthesis

HW synthesis has been explored for many years and good algorithms are available for generating net lists of gates from an RTL description of the logic to implement. In our vision for system level design, if the modules to be generated refer to an implementation fabric made of gates or low complexity logic, we can apply existing tools to the task. For example, if the implementation fabric is an FPGA block or a standard cell module, we can use a number of commercially available solutions. However, if the hardware architecture of the implementation module is not specified, we are in the general domain of behavioural synthesis that, although many researchers have dedicated the best of their time to the topic, has not seen a strong endorsement to put it mildly in the commercial domain. By leveraging an understanding of platform-based design it is possible to advance the state of the art in this domain

Yet, another interesting topic is the mapping onto reconfigurable hardware such as reconfigurable processors as they are making inroads in the IP market place. For example, ARC cores and Tensilica are indeed capturing the attention of several companies who see an advantage in being able to customise instruction sets and micro-architectures of programmable elements. While tools

are available from vendors, we believe that the state of the art here is still primitive. For example, there is also a strong interest in generating VLIW processors on the fly or dynamically reconfiguring them while executing code. Indeed, we see that the choice of implementation fabrics is increasingly varied with the exploration of almost any intermediate point in the architectural space between fully programmable cores and custom hardware. For example, the HP labs PICO project on automatic generation of VLIW processors together with the optimised compilers for the selected structure is very interesting both from the academic and the industrial point of view [11]. The work in the Mescal project of the GSRC, carried out in collaboration between Berkeley and Princeton is in this generic research area [12]. Given our interest in platforms and their evolution, we propose to track this work with great attention. Once more, the collaboration with our industrial partners will be invaluable to judge the promises of the various approaches and the gaps that we need to fill to make hardware reconfigurability a reality.

When we think of hardware synthesis, with an emphasis on reuse, we cannot overemphasise the importance of the communication mechanisms to be selected and of the interfaces among the components of the design. Thus far, the common approach is to standardise hardware interfaces, towards a common interconnection mechanism (for example a bus with a given architecture and arbitration protocol).

In our vision, the interconnection pattern has to be as free as possible since much there is to gain by leveraging degrees of freedom in communication. However, exploiting these degrees of freedom is not straightforward; this implies that the interfaces cannot be standardised but must be generic. In our concept, there is a clear path to ease the selection process of the interconnection patterns, and the related interface generation. Indeed, we believe that interfaces can be automatically synthesised very effectively.

4.7 Verification

In this approach, mathematically rigorous and checkable refinement methods from well-understood specification models should be used. In this area, we believe that using formal specifications will make the application of formal methods easier and will eliminate the need for verifying properties that are satisfied by construction. In addition, verifying whether an implementation satisfies the original specification can be made much easier if formal successive refinement techniques are used. Finally, the adoption of automatic synthesis can reduce the need for implementation verification to a point where formal equivalence checking technique will be feasible between several abstraction layers. However, since there is much not yet known in this field (for example, how much scope of real design will the formal models cover?), we must be prepared to mix semi-formal, simulation/analysis/assertion-based checking in with the formal methods.

5. CONCLUSION

In this paper, we have surveyed the problems of developing embedded SW for modern products, which are considerable. However, merely listing the problems and issues is not enough. We have also outlined a clear vision of the kind of embedded software development methodology that is possible in the future. Such a vision cannot be implemented with today's tools, and state of the art research. In addition to painting this vision, we have

outlined key areas for research that are required to allow it to be realised and some approaches we have followed in our work at UC Berkeley and Cadence Research Laboratories. Although visionary, we believe that pragmatic progress will be possible and the next few years will see substantive steps being taken towards realising our goals.

6. ACKNOWLEDGMENTS

We would like to acknowledge, with thanks, ideas and suggestions for this vision from Alberto Ferrari, Felice Balarin, Jerry Burch, Tom Henzinger, Ed Lee, Luciano Lavagno, Lane Lewis, Richard Newton, Roberto Passerone, Ellen Sentovich, Frank Schirrmester, Marco Sgroi, Ted Vucurevich, and Yoshi Watanabe.

7. REFERENCES

- [1] A. Sangiovanni-Vincentelli and A. Ferrari, "System Design - Traditional Concepts and New Paradigms", ICCD 99, Austin, October, 1999, 2-12.
- [2] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly and L. Todd, *Surviving the SOC Revolution: A Guide to Platform-Based Design*, Kluwer Academic Publishers: November, 1999.
- [3] G. Martin, "Productivity in VC Reuse: Linking SOC platforms to abstract systems design methodology", *Forum on Design Languages: Virtual Components Design and Reuse*, Lyon, France, August-September, 1999, 313-322.
- [4] "Technology Roadmap on Software-Intensive Systems: The Vision of ITEA (SOFTEC Project)", issued by the ITEA Office, Eindhoven, March 2001. Chapter 7, "Engineering", 47-56.
- [5] G. Martin, L. Lavagno, J. Louis-Guerin, "Embedded UML: a merger of real-time UML and co-design", *CODES 2001*, Copenhagen, April 2001, 23-28.
- [6] P. Alexander, R. Karnath, and D. Barton, "System specification in Rosetta", *Engineering of Computer Based Systems -ECBS 2000*, 299-307.
- [7] F. Balarin, "Worst-case analysis of discrete systems", *ICCAD 1999*, 347-352.
- [8] S. Swan, "An Introduction to System-Level Modeling in SystemC 2.0", white paper on the Open SystemC web site, 2001, http://www.systemc.org/papers/SystemC_WP20.pdf
- [9] Marco Sgroi, Luciano Lavagno, Alberto Sangiovanni-Vincentelli. *Formal Models for Embedded System Design*. *IEEE Design & Test of Computers*, 17(2), June, 2000, 14-27.
- [10] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*, Kluwer Academic Publishers: 1997.
- [11] S. Aditya, B. Ramakrishna Rau, and V. Kathail, "Automatic architectural synthesis of VLIW and EPIC processors", *ISSS 1999*, November, 1999, 107-113.
- [12] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey and A. Sangiovanni-Vincentelli. *System Level Design: Orthogonalization of Concerns and Platform-Based Design*, *IEEE Trans on CAD of Integrated Circuits and Systems*, 19(12), December 2000, 1523-1543.