System Synthesis for Multiprocessor Embedded Applications

Luigi Carro[*] Márcio Kreutz[+] Flávio R. Wagner[+] Márcio Oyamada[+]
[*]DELET – UFRGS
Av. Osvaldo Aranha, 103
90035-190 Porto Alegre – RS Brazil
carro@iee.ufrgs.br

[+]PPGC – UFRGS
Bento Gonçalves, 9500 Bloco IV – CP 15064
91501-970 Porto Alegre – RS Brazil
{ kreutz, flavio, oyamada}@inf.ufrgs.br

Abstract

This paper presents the system synthesis techniques available in S^3E^2S, a CAD environment for the specification, simulation, and synthesis of embedded electronic systems that can be modeled as a combination of analog parts, digital hardware, and software. S^3E^2S is based on a distributed, object-oriented system model, where objects are initially modeled by their abstract behavior and may be later refined into digital or analog hardware and software. System synthesis is targeted to a multiprocessor platform. Each processor, either a custom-designed one or an off-the-shelf component, can have a specialized behavior, like signal processing or control processing. The environment selects processors that best match the desired application by analyzing and comparing processor and application characteristics. The paper illustrates the architecture selection process with concrete examples.

1. Introduction

Presently, the behavior of a complete electronic system can hardly be classified as control dominated or data dominated. The current trend is to have a mix of behaviors in the same system-on-a-chip, requiring a combination of different design styles. Typical examples of such systems are portable multimedia devices, industrial distributed controllers, and vehicle supervision systems. All these systems demand digital signal processing, analog circuits to interface with the real world, radio frequency communication links and scalar processing (for database lists, display and keyboard control). These systems are a major trend in the system industry, and their design is currently performed using a mix of different methodologies.

In the S^3E^2S (Specification, Simulation, and Synthesis of Embedded Electronic Systems) design environment, complex electronic systems can be modeled as a combination of objects described at different abstraction levels and different domains – abstract behavior (expressed by a high-level, object-oriented specification), digital hardware, analog hardware, and software. By coupling different simulation engines [1], the environment supports co-simulation of any multi-domain description obtained during a process of stepwise design refinement.

The synthesis step is not targeted at a single, specific processor architecture. S^3E^2S allows an easy design space exploration at the multiprocessor level, whereby different processor architectures are analyzed and those best matching the desired application are selected and combined for design refinement. S^3E^2S tries to use available processors as much as possible, in order to reduce system costs in terms of hardware and to enhance design time. Since nowadays one can find different microprocessors offering different trade-offs between cost, architecture and power consumption, their use in the design cycle generally turns out to be a flexible and low cost solution. Moreover, one must consider that designs seldom are started from scratch. Most companies try to reuse previously designed boards, multi-chip modules or IP processors, for which they additionally have a library of available software modules. Small and medium companies rarely have the needed capital to invest in a high volume, single chip solution, so using programmable processors is a natural choice for starting the design of a new product.

The combination of multi-domain modeling and co-simulation with multiprocessor synthesis based on automatic processor selection is a distinguished feature of S^3E^2S when compared to other environments for the design of embedded systems.

This paper is organized as follows. The next section presents a comparison of S^3E^2S with other design approaches. Section 3 presents a very brief overview of the modeling and co-simulation capabilities of S^3E^2S, followed in Section 4 by the methodology for the processor selection and the hardware synthesis techniques used in the environment. Section 5 presents case studies that illustrate the versatility of the design environment. Section 6 draws conclusions and discusses future work.

2. Comparison with related work

The specification and simulation of application-specific embedded systems is an area of active research. In the case of complex systems, which cannot be implemented by a single processor or controller and its associated software, it is difficult to specify the designer's intention. Many specification languages, or combinations of languages, are being used in industry [2].

The description of complex systems through a single, abstract language has been proposed [3,4]. Some approaches that follow this strategy adopt an object-oriented specification to describe both hardware and software [5,6]. In these cases, design partitioning is left to later design stages. The system is modeled as a set of

objects, and each one of them may be later implemented as software or hardware, either digital or analog (as in [7]).

An alternative approach supports modeling of heterogeneous systems. Ptolemy [8], for instance, is an environment for simulation and prototyping of heterogeneous systems, which also uses object-oriented technology. Ptolemy implements the combination of different simulation mechanisms, called domains (such as Synchronous Data Flow, Dynamic Data Flow, discrete event, and analog). Another environment allowing the specification and simulation of heterogeneous systems is described in [9], where a backbone in the operating system implements communication among dedicated simulators that are needed for heterogeneous objects specified in different languages.

S^3E^2S combines the advantages of the multi-language and heterogeneous simulation approach with the abstract, object-oriented specification. It offers an object-oriented modeling environment, where all objects can be modeled regardless of their future implementation as digital or analog hardware or software. Then, any object may be specified in any of these domains or refined into any of them, and every possible intermediate model during this stepwise refinement process may be co-simulated.

Most work on hardware and software co-design focuses on the synthesis of a dedicated hardware or of a dedicated instruction set processor [3,6,8,10-13]. In the Polis system, for instance, the target architecture is a set of commercial processors [14]. However, all processors have the same characteristics. They are microcontrollers, targeted to embedded control and not to data intensive applications. The synthesis style is based on software synthesis and performance estimation techniques. In S^3E^2S, we also aim at using as much software as possible, in order to reduce system cost and design time. Differently from Polis, however, our target system can contain data-dominated and control-dominated behavior, and the system tries to find the best processor (according to some design criteria) for each task.

The S^3E^2S environment focuses on a multiprocessor paradigm. Instead of having a fixed target architecture devoted to ASICs or ASIPs, synthesis in S^3E^2S is based on a library of processors, each with different characteristics, ranging from microcontrollers to digital signal processing machines, with different architectures available in each domain. Each object of the specification may be mapped to a single processor, and each processor may execute the function of one or more objects, as it will be shown in Section 4.

3. Modeling applications

S^3E^2S is built on top of SIMOO [15], an integrated environment for object-oriented modeling and simulation of discrete systems. SIMOO is composed of a class library and a model editor. The editor supports the description of the static and dynamic aspects of the model. The static structure is described graphically, while the dynamic structure is described either directly in C++ using the library resources or by means of a state diagram annotated with C++ code. The editor implements extensions to diagrams usually proposed by object-oriented design methodologies, in order to handle simulation-related aspects. From the model description, the editor automatically generates the necessary executable code.

A model is composed of interface elements, which support user interaction, and autonomous elements, that are used to model concrete entities. An autonomous element is an active object, i.e., an object with its own execution thread and a message queue. It may interact with other autonomous and interface elements only through messages. The model does not support shared variables, so that it may be also used in distributed environments. Different objects of the same model may follow different *paradigms*. A paradigm is defined as a combination of the following modeling approaches: event orientation or process orientation for the description of the object behavior, messages or ports for the communication between objects, and active or passive message handling. These approaches may be extended or specialized by inheritance.

More details on the modeling possibilities of S^3E^2S and on its mechanisms for co-simulating mixed analog, digital and software descriptions can be found in [1].

4. Object evaluation and processor selection

After modeling the system in the object-oriented environment, the primary goal to obtain a working system is to map objects to one or more physical processors. This strategy assumes that, nowadays, there are different commercial processors available, with different cost/performance ratios, ranging from high performance DSPs to low power microcontrollers. If the user can find a set of commercial processors that implements the desired behavior, design costs would go down, for core-based designs can favorably replace crude hardware synthesis, at least for small or medium volumes or for fast prototyping with small costs. For this reason, S^3E^2S first tries to find, from a processor library, a set of possible candidates that can implement the desired behavior.

This evaluation phase must be performed as soon as possible in the design cycle, but at the same time without the burden of extensive code simulations. The goal is to find a set of processors that, from the abstract object specification, can implement the desired behavior. Compiling the original code to specific processors and then simulating or emulating the processor is a time consuming task. If a high level evaluation is possible, the designer can try different modeling styles of the design in order to find the best set of processors. For example, one might try to reuse a board or a SOC with different processors like a microcontroller and a DSP plus some analog interface already available. Alternatively, one

might ask which specific set of processors could be used in a new development.

4.1 Object evaluation

In [16], the evaluation of software performance is based on a two step procedure. Firstly, a high- level processor-independent representation is obtained, like a CDFG (control and data flow graph), and then the CDFG is translated into C code for the target processor. In S^3E^2S we also use an intermediate description of the code to be executed. Some optimizing transformations are executed, and upon this optimized CDFG structure an evaluation of the object behavior is obtained. Differently from [16], however, whose work is targeted to controllers, each software module in S^3E^2S is free from any previous template, and so each object has any possible behavior. This way, one must characterize the typical behavior of the object code, among three alternatives: a) control dominated, as in FSMs for controllers; b) data intensive computations, as in digital filters; or c) memory intensive computations, as in list processing or data-base searching in a building entrance control, for example.

Each object is targeted to a processor that best implements its behavior. The criteria for choosing the best processor are based on the processor characteristics to execute the desired code. For example, a DSP processor with a deep pipeline will pay a high branch penalty and is thus not adapted to a control-intensive application. On the other hand, if a low cost microcontroller can be used in a slow varying process that requires digital filtering at largely spaced samples, then this solution should also be given as an option for the designer.

There are different target architectures available in S^3E^2S. Either a full synthesizable processor might be generated, or a set of heterogeneous processors might be required. For example, a DSP processor and a small microcontroller, both commercially available, could be used together in a certain application. Eventually, whenever none of the available processors can be selected, due to execution time limitations or other factors, a dedicated integrated circuit might be developed.

After parsing the application, a CDFG is obtained, and in the sequence a machine-independent 3-address code is generated. In order to better analyze each processor, three types of virtual machines were defined, according to different target architectures, like microcontroller, RISC and DSP.

Each one of the virtual machines has particular characteristics. The Microcontroller-like virtual machine is defined as an instruction set in which the target architecture has only two registers. The RISC virtual machine has as many registers as necessary to allocate all data dependent variables. These variables are those that are called temporary symbols in a three-address code, because they hold values resulting from one operation and will be used in another subsequent operation. This RISC virtual machine type characteristic allows the optimization of memory accesses by the compiler. Finally, the DSP virtual machine type has as many registers as the number of the variables used in the application, so that variables are accessed in memory only once. This is to reflect the large available internal memory and the short access time of this type of architecture. Also, the multiply and accumulate (mac) instruction is detected, and when it occurs in the application code, only one cycle is computed for data transform operations.

The next step concerns object analysis, where the dominant characteristic of the object is identified: control intensive (many control instructions and flow breaks), memory intensive (list processing, digital filtering, much memory usage) or data processing intensive (few memory accesses, most processing done with internal registers). Each one of these characteristics will favor a different processor in the library.

Let

- M be the total number of cycles used in memory access in the internal 3-address code,
- P the number of cycles to execute all data transformations (add, sub, and, mult, etc.), and
- C the total number of cycles taken to test and branch (control instructions).

Let

- APb (application profile) be the relative importance of each behavior “b” in comparison with others, expressed as

$$APP = P / (P + M + C), \quad (1)$$
$$APM = M / (P + M + C), \quad (2)$$
$$APC = C / (P + M + C). \quad (3)$$

Equations 1 to 3 show the relative importance of improving a certain architecture to obtain the maximum gain while executing the function of the particular object. This way, if an application has an APC of 0.7, this means that it is control-dominated, and there is no point in using a DSP processor to implement it [17].

The serialization of functions can be tried, so that a group of objects can be mapped to a single processor and the application may fit in a smaller number of processors. At the beginning of this allocation process, all actions that the user required to be executed in parallel will be necessarily allocated to different processors. Regarding other actions, the communication protocol is checked. In case of synchronous communication, actions of the communicating objects are naturally sequential and may be allocated to the same processor.

4.2 Processor analysis and selection

In order to be able to find the best processor for a given object or group of objects, we also need to characterize processors. This characterization is performed once for each object and stored in the library, together with the processor description itself. The characteristics listed in Table 1 provide a high-level abstraction of a processor from a behavioral point of

view. They can also be used to classify application-specific processors, like those devoted to DSP. These processors will have some peculiarities in operand addressing (for example, circular buffering), the number of busses to access memory, type of control instructions (easy definition of loops at the assembly level), and depth of pipeline.

Table 1 – Processor characteristics

Size of binary word
Types of instructions
Memory operand accessing modes
Execution time of each instruction, in clock cycles
Number of busses to access memory
Type of memory
Number of registers
Control instructions
Use of pipeline and depth of eventual pipeline
Use of Harvard architecture or not

Consider a particular application, for which APP, APM and APC have been calculated. One must now evaluate the cost of using a given processor for implementing this application. Considering the previous characterization of this processor, its Application Performance Distance (APD) is obtained by the following distance measure:

$$APD = \sqrt{(Pi - APP)^2 + (Mi - APM)^2 + (Ci - APC)^2} \quad (4),$$

where index i stands for a certain processor, and P_i, M_i and C_i are the relative costs of the processor instructions to execute data transformations, memory accesses and control operations, respectively. Each processor obtains the Pi, Mi and Ci values from its own virtual machine. For this analysis, it is assumed that each 3-address code instruction generates a single instruction in the target processor. Equation 4 gives a clue on the performance of this processor when executing the given application.

The 3 dimensional distance measurement obtained by using equation 4 shows how different the processor is from the ideal virtual machine that can execute the code. The processor with the smallest distance will be probably best suited to execute the application with less overhead.

5. Case Studies

5.1 Initial examples

Three very different processors have already been characterized in the S^3E^2S library: the 8051 microcontroller [18], the C25 digital signal processor [19], and the RISCO microprocessor, a 32-bit RISC-like microcontroller [17]. Table 2 shows some of the processors' characteristics. For example, since the C25 has a DSP architecture, memory accesses and computations take the same amount of cycles. This favors data intensive applications. On the other hand, a RISC machine with many registers favors computations with few memory accesses. At the same time, the cost of a branch is higher in the C25, thanks to the effect of the possible pipeline flush. The added cost of the flush is considered in table 2. In the C25, internal memory is considered as a register bank, due to its small access time and to the special indexing registers available in the architecture.

Table 2 – Partial processor characterization

	# of registers	Jump cycles	Memory access cycles	# of busses
8051	8	2	2	1
RISCO	32	1+2	2	1
C25	544	4+2	1(internal)	2

In order to illustrate the processor selection strategy, we take 3 different algorithms, presented in Figure 1. The computation of the distance a person walks or runs in the PODOS system[1] is shown in Figure 1(a). We also have a digital filter, expressed in Figure 1(b), and a simple dot product, shown in Figure 1(c).

The result of the analysis of the three algorithms is presented in Table 3. Each algorithm was classified according to its main characteristic, as explained in Section 4.2. As shown in Table 3, the PODOS integration algorithm is mainly memory intensive (APM=0.523), the filter is mainly data intensive (APP=0.538), and the dot product is also mainly data intensive (APP=0.484). The resulting Performance Profiles for each application and processor can be found in Table 4.

```
for(int i; i < n; i++) {
   vel1= sqrt((acel_v*acel_v) + (acel_h*acel_h));
   mean = (((vel_prev + vel1)/2) * Tsampling);
   vel_prev= vel1;
   vel = mean;
}
```
(a) Podos Integration

```
while (i < I) {
     aux= x[i] - y[i];
     aux= aux * K;
     y[i+1]= y[i] + aux;
     i++;
}
```
(b) Digital filter

```
prod= i= 0;
while(i < Array_Size) {
     prod= prod + a[i] * b[i];  i++;
}
```
(c) Dot Product

Figure 1 – Example algorithms

[1] The PODOS system is an integrated circuit that measures the distance a person walks or runs. It is placed on the shoe and communicates with a display on the person's wrist.

Table 3 – Evaluation of Application Profiles for the PODOS example algorithm for all processors

	APC	APM	APP
8051	0.038	0.615	0.346
Risco	0.047	0.523	0.428
C25	0.090	0.090	0.818

Table 4 – Evaluation of Application Performance Distance for the example algorithms and processors

	APD		
	8051	RISCO	C25
PODOS	0.639	0.313	0.353
Filter	0.860	0.156	0.092
Dot	0.924	0.258	0.207

The system then suggests the best processor, but allows the use of another one, in case the designer wants to reuse some module or a previously designed board. For the given applications, the following results are achieved:

- PODOS integration (memory-intensive): the processor with smallest APD for this algorithm is the Risco, since the mixed behavior favors a general-purpose architecture;
- Filter (data-intensive): the processor with smallest APD for this algorithm is the C25.
- Dot product (data-intensive): The processor with smallest APD for this algorithm is again the C25.

5.2 Crane control

In order to illustrate more design possibilities using S^3E^2S, a more complex example has been modeled. This system, composed of a crane and its embedded control, has been proposed in [20] as an attempt of benchmarking in the area of system-level modeling and synthesis.

The physical plant is composed of a crane with a load, moving along a track, as depicted in Figure 2. The modeling of the physical system is done by a set of differential equations, which describe the behavior of the crane with a load and external forces being applied. The control of the system involves a set of sequential procedures and the control algorithm itself, which will assure a smooth behavior while the car is moving.

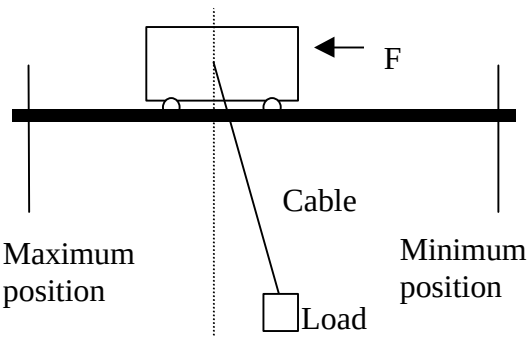

Figure 2 - Crane moving along its track with load [20]

The control algorithm is implemented as a discrete computation of the state-variable method. In the control algorithm, the goal is to move the crane with a linear displacement, without bumps and oscillations. A set of matrix multiplications must be performed at a fixed time step of 10 ms. If $q_n = [q1_n, q2_n, q3_n, q4_n, q5_n]^T$ is the discrete state vector of the crane, then

$q_{n+1} = A*q_n + B*[Motor\_Voltage \;\; Car\_Position]^T$ (7)

is the next discrete state of the control algorithm. Coefficient matrixes A and B have dimensions 5x5 and 2x5, respectively.

The control algorithm outputs the value of the force to be applied to the crane, and this is passed to other objects, that are responsible for driving the dc motor that controls the speed, the breaks, and the emergency break (that completely stops the crane until a power-on-reset is performed). A more detailed description of this modeling can be found elsewhere [1]. In the following, we consider only part of the system behavior, concerning the crane control and movement, which is the only synthesizable part of the modeling.

In a first modeling, the system has been partitioned into 4 objects, one responsible for the finite state machine of the whole controller, and three other objects for performing different multiplications regarding matrix operations to compute position and the forces to be applied. Table 5 shows the Application Performance Distance of the three processors of the library for this application.

Table 5 – Application Performance Distance for the processor with regard to the crane control, first modeling

Objects	Processor best APD		
	8051	RISCO	C25
Control	0.633	0.271	0.180
Mul_aq	0.754	0.176	0.230
Mul_y	0.917	0.177	0.195
Mul_bx	0.753	0.208	0.235

Although the Control object is not mainly data intensive, the C25 was chosen because the combination of its instruction set with the used virtual machine showed the smallest distance. This means that the compiler that will effective generate code for this application will use its characteristics in a better way, and the outcome will be more predictable. The three remaining objects, that perform the matrix multiplications, are more operation intensive than memory intensive (matrixes are small) and the Risco processor is a better match.

In a second modeling, the function of all former four objects is implemented by a single object. Table 6 shows the Application Performance Distance in this case. The application is mixed, and so the system chooses the Risco processor as the best solution.

Table 6 – Application Performance Distance for the crane control, second modeling

APD/Proc	8051	Risco	C25
Crane	0.658	0.187	0.240

As it can be seen, S^3E^2S can not only guide the design process, but it can also help the designer in the

specification phase for purchasing an IP or in the development of a new architecture. After processor selection, the C code for the dedicated processor is generated, and a dedicated commercial compiler is used to obtain the final object code.

6. Conclusions and future work

The automatic design of embedded electronic systems is an open area of research. An integrated CAD environment must consider important aspects such as system specification, validation, and synthesis. Various different approaches have been proposed in the literature to cope with these issues.

For the synthesis of embedded systems, most CAD environments have a fixed target architecture, consisting of a single processor and maybe some peripheral ASICs. These synthesis approaches concentrate on the task of partitioning system functions among hardware and software. S^3E^2S, in turn, performs a synthesis that is based on a library of processors, ranging from microcontrollers to ASIPs and DSPs. Each processor is characterized by a set of parameters, and the environment tries to match each object of the application (considering the application profile) to the most adequate processor. The final architecture is therefore a multiprocessor platform. The paper presented examples illustrating the utilization of the design infrastructure described above, where the selection of the processors is performed in an automatic way.

In all shown examples the microcontroller (8051) has never been chosen. This is because the use of a microcontroller is not related with performance, but with less area or lower power dissipation. This way, we are currently investigating a more refined processor selection procedure, where other design related issues like area and power can be also evaluated. Moreover, some fuzzy design criteria like availability in the market or of trained designers with a particular architecture will also be included in the processor selection metrics.

We are also currently investigating the communication costs between processors and a strategy for automating the selection of the best communication infrastructure. Future work also includes the expansion of the processor library and the development of algorithms to automatically group objects into a single processor.

References

[1] Wagner, F.; Oyamada, M.; Kreutz, M.; Carro, L. “Object-Oriented Modeling and Co-Simulation of Embedded Systems”. In: IFIP X International Conference on VLSI, Lisbon, December 1999, p.439-446.

[2] W.Nebel and G.Gorla. “JAVA, VHDL-AMS, ADA or C for System Level Specification?” In: Design, Automation and Test in Europe, 1999. Proceedings**,** IEEE Computer Society Press, 1999. p 720.

[3] J.K.Adams and D.E.Thomas. “The Design of Mixed Hardware/Software Systems”. In: ACM/IEEE Design Automation Conference, 1996. Proceedings, ACM Press, 1996. pp 515-521.

[4] G.Martin. “Design Methodologies”. In: Design, Automation and Test in Europe, 1998. Proceedings, IEEE Computer Society Press, 1998. pp 286-289.

[5] J.Böttger et al. “An Object-Oriented Model for Specification, Prototyping, Implementation and Reuse”. In: Design, Automation and Test in Europe, 1998. Proceedings, IEEE Computer Society Press, 1998. pp 303-310.

[6] W.Wolf. “An Architectural Co-Synthesis Algorithm for Distributed, Embedded Computing Systems”. IEEE Transactions on VLSI Systems, June 1997. pp 218-229.

[7] J.Z.Pino and K.Kalbasi. “Cosimulating Synchronous DSP Applications with Analog RF Circuits”. In: 32nd Asilomar Conference on Signals, Systems and Computers, 1998.

[8] A.Kalavade and E.Lee. “Hardware/Software Codesign Using Ptolemy”. In: Codesign. Edited by J.Rozenblit and K.Buchenrieder. IEEE Press, 1995. pp 397-413.

[9] A.A.Jerraya and R.Ernst. “Multi-language System Design”. In: Design, Automation and Test in Europe, 1999. Proceedings, IEEE Computer Society Press, 1999. pp 696-699.

[10] M.Mrva, K.Buchenrieder and R.Kress. “A Scalable Architecture for Multi-Threaded Java Applications”. In: Design, Automation and Test in Europe, 1998. Proceedings, IEEE Computer Society Press, 1998. pp 868-874.

[11] P.Schaumont et al. “A Programming Environment for the Design of Complex High Speed ASICs”. In: Design Automation Conference, 1998. Proceedings, ACM Press, 1998. pp 315-320.

[12] E.Berrebi et al. “Combined Control Flow Dominated and Data Flow Dominated High Level Synthesis”. In: ACM/IEEE Design Automation Conference, 1996. Proceedings, ACM Press, 1996. pp 573-578.

[13] L.Freund et al. “A Codesign Experiment in Acoustic Echo Cancellation: GMDFα” In: IEEE International Conference on System Synthesis, 1996. Proceedings, IEEE Computer Science Press, 1996. pp 83-89.

[14] M.Chiodo et al. “Hardware Software Codesign of Embedded Systems”. IEEE Micro, August 1994. pp 26-36.

[15] Copstein, B.; Wagner, F; Pereira, C. “SIMOO – An Environment for the Object-Oriented Discrete Simulation”. In: 9th European Simulation Symposium, 1997. Proceedings, SCS, 1997. pp 21-25.

[16] K.Suzuki and A.Sangiovanni-Vincentelli. “Efficient Software Performance Estimation Methods for Hardware/ Software Codesign”. In: ACM/IEEE Design Automation Conference, 1996. Proceedings, ACM Press, 1996. pp 605-611.

[17] Alba, C.; Carro, L.; Suzim, A. “Embedded Systems Design with Frontend Compilers”. In: IEEE International Conference on Computer Design, 1996. Proceedings, IEEE Computer Society Press, 1996. pp 200-205.

[18] Intel. “Microsystems Components Handbook”. Volume I, 1985.

[19] Texas Instruments. “TMS320Cxx User´s Guide”. 1997.

[20] E.Moser and W.Nebel. “Case Study: System Model of Crane and Embedded Control”. In: Design, Automation and Test in Europe, 1999. Proceedings, IEEE Computer Society Press, 1999. pp 721-723.