An Analysis of Hardware/Software Co-Design Architectures Using SAAM

Kamran Sartipi
Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1
E-mail: ksartipi@neumann.uwaterloo.ca

Abstract. This paper presents a brief analysis of some approaches in the field of Hardware/Software (HW/SW) Co-Design. The main efforts in the design of these systems are focused on the HW/SW partitioning process, and little work is devoted to providing a modern system development environment in this field. In the design process, some changes to the architectures of the systems are needed to adopt these enhancements. In order to investigate and evaluate different architectures in this regard, i.e. non-functional qualities, an analysis method is required. SAAM, originally devised for the analysis of software architectures, is utilized to compare and evaluate three co-design systems, based on different task-scenarios and functional decompositions.

Keywords. Software Architecture; Software Architecture Analysis Method; Hardware/Software Co-Design; Candidate Architectures; Applications of Scenarios; Evaluation of different Architectures.

1 Introduction

Hardware/software co-design addresses the design of an *embedded* system using both hardware and software components. This research area is increasingly important since the embedded systems are becoming more complex and requiring better cost-performance achievement. Co-design is applicable to a range of systems from small home appliances to large real time systems.

The key task in HW/SW co-design is the automatic partitioning of the system specification into hardware and software parts. Partitioning algorithm uses specific criteria to perform the partitioning task. The algorithm searches through the system

specification, which is usually converted to a graph representation, to extract the partitioning related information. As a simple rule, parallelism is a criterion for hardware implementation, sequential execution and behavior control are criteria for software implementation. User defined parameters concerning area i.e. hardware elements in a chip or the code used for software, timing, etc., build the framework for the evaluation of the partitioned system. These parameters constitute the components of a function, known as *cost-function*, which evaluates the performance of the partitioned system.

In most approaches there is a predefined target architecture consisting of a processor which executes the software part, and a set of co-processors which implement the hardware functions of the partitioned system. Other configurations are also used. Software compilation and hardware synthesis are required tasks to realize the partitioned system into target architecture.

The co-design environment can be viewed as a combination of diverse design tools which are collected in one place. Some co-design approaches rely on the general design tools to perform parts of their design tasks. General tools such as high level hardware-synthesis system (*OLYMPUS*) [3], and Parallel-Virtual-Machine (PVM) [4] for system simulations and verification, are available as support tools.

Existence of efficient design tools in well established environments such as *CASE* in the software-system domain, and *CAE* in the hardware-system domain, motivates applying the same techniques to the co-design domain which includes ideas from both areas. Available tools in *CASE* and *CAE* provide an environment for rapid and interactive software construction, testing and maintenance. These tools are intended to serve a team of designers to work on a large project. In some cases different design tools are gathered in one package namely integrated development environment to provide a homogeneous and friendly appearance of the whole system. *Statemate* tool[1], *SPeeDCHART*[2], and *ExpessV-HDL*[3], are examples of this kind. There are also environments for tool integration which facilitate the interface and communication among individual design tools in that environment. *HP SoftBench* [5] is a member of this group. The above tools in both domains (software and hardware) are attributed with some major characteristics such as: Use of graphical specification techniques, simulation of specification, code generation in common programming languages, use of project management, document generation, etc. These characteristics, are also feasible for an integrated HW/SW co-design tool.

We are interested in analyzing successful co-design systems to evaluate their flexibility for adoption of some changes in their architectures. We hope this anal-

[1] *Statemate* is a registered trademark of i-logics inc.

[2] *SPeeDCHART* is a trademark of Speed S.A

[3] *ExpessV-HDL* is a registered trademark of i-logics inc.

ysis to assist us in designing a proper architecture for the co-design system we are working on. We use an analysis method, designed for evaluation of non-functional qualities (modifiability, scalability, security, etc.) of software system architectures, for the co-design systems. The method is called SAAM [1] which is stand for Software Architecture Analysis Method. A brief description of the SAAM method is also available in section 5. This paper has one main goal which is the evaluation of the selected co-design architectures, and one side effect which is the test of applicability of SAAM, originally devised for software architectures, on a different field. The SAAM method has been successfully applied to a few important software system domains such as Internet Information Systems (IIS) [2], and User Interface domain [1].

This paper is structured as follows: The next three sections briefly describe common HW/SW co-design approaches namely, *UNITY*, *COSYMA*, and *CASTLE* with respect to their system architectures. In section 5 The *SAAM* method is described and the notion of *task scenarios* as a basis for functional partitioning and architectural evaluation are presented. The above co-design approaches are then evaluated using SAAM method and a comparison table is presented. Finally section 9 gives a conclusion to the paper. Before starting the description of the co-design systems, it

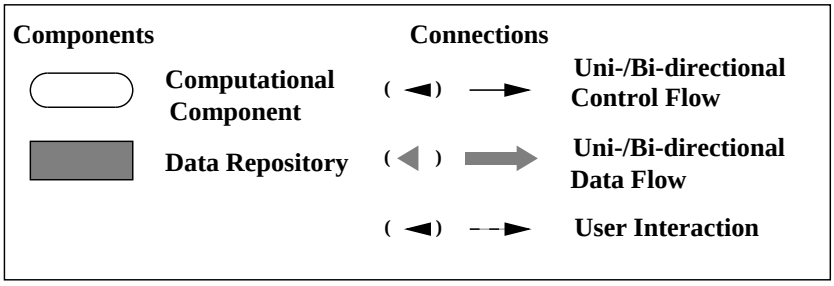

Figure 1: Common structural notation used to represent the co-design architectures.

is important to note that the architectural notation presented here is different from the notation used in original papers. In order to analyze and compare a number of independently developed systems, it is helpful to first represent these systems with a common structural notation. The notations are shown in Figure 1.

2 The UNITY system

The *UNITY* approach to HW/SW co-design [8] employs an automatic partitioning algorithm which separates the system specification into hardware and software, independent of the synthesis method applied for the target system. *UNITY* exploits of

two-stage clustering technique to partition the unity-elements (the smallest unit in a *UNITY* program), into hardware and software parts. The architecture of the *UNITY*

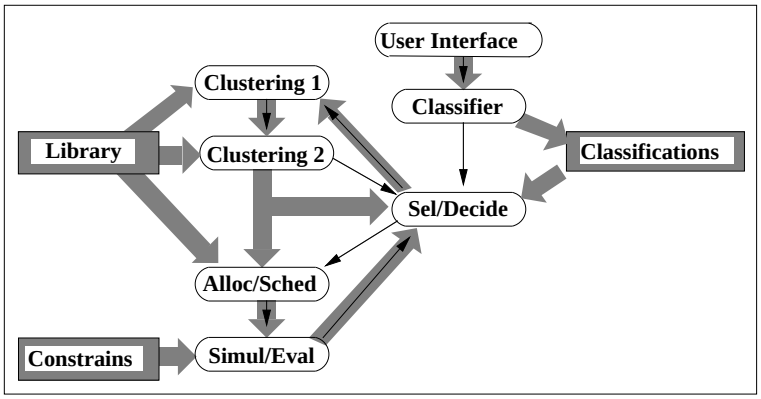

Figure 2: The architecture of the *UNITY* approach.

approach is shown in Figure 2. In *UNITY*, the system specification is defined in the *user interface* unit which is then parsed and an abstract syntax tree is produced. The task of *Classifier* unit is to verify the relationship among *UNITY* elements in the parsed tree with regard to some defined attributes, and to build a classification table of different implementation alternatives. This table is kept in a data repository called *classifications*. The *Sel/Decide* unit selects a particular implementation alternative as reference for the clustering process, and dispatches it for the partitioning procedure. Partitioning consists of two dependent clustering procedures. At first stage (*clustering*

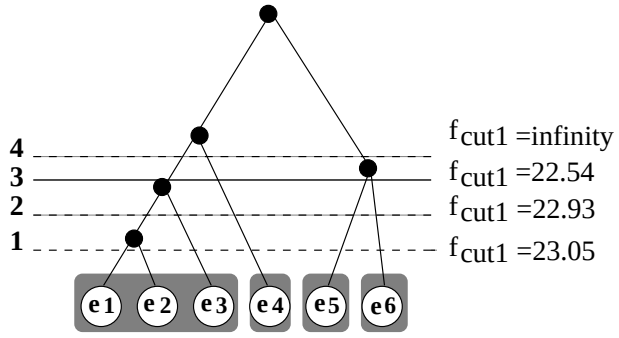

Figure 3: The cluster tree, builds clusters from elements e1 to e6.

1), the elements are grouped (clustered) according to their implementation similarities and a cluster tree is set up (Figure 3). At the next step of this stage, the cluster tree is cut at some layer, with the goal of minimizing the cost function. The cutting algorithm compares different cut lines to find the one with minimum cost. The result of the cut line at the first stage is a set of subtrees which is used as the inputs for the second clustering stage. The algorithm for the second stage clustering (*clustering 2*) is similar to that of the first stage with different criteria. A *library*, containing the attribute lists of the available complex components also exists. This library is used by the clustering algorithms to group the elements having commonalities according to the applied criteria.

If the current gathering of elements in clusters and the result of the cost function is not satisfactory, The *select/decide* unit selects another implementation alternative from the available set in the classification table and the partition procedure is repeated. This cycle is iterated until one of the alternatives is selected. The user interaction or the limited number of iterations prevents the free running of the partitioning loop. When an implementation of clusters is acceptable, it is fed to *allocation/schedule* unit and the allocation process starts. It assigns each cluster to a separate module of a component in the library.

In order to design the communication interface and control sequence among the modules in software and hardware, a particular graph called *interface graph* is developed by the partitioning unit.

The *scheduling* process searches through the interface-graph and establishes the activation sequence among the created clusters. The *interface graph* and the target system is then used by the *simulate/evaluate* unit to assess the target system. The simulation is performed using a general tool (parallel virtual machine). The evaluation of the created system is based on the comparison of simulation results against the design constraints. The result of the evaluation is then used by the *select/decide* unit to determine if the final system is acceptable within the user defined constraints or not. In case of unsatisfactory results, the whole design procedure is repeated by selecting another implementation alternative from the classification table.

3 The COSYMA system

COSYMA (CO-SYnthesis for eMbedded Architecture) [7] is an automated hardware/software partitioning system for co-designing of small embedded systems such as micro-controllers. The architecture of *COSYMA* is shown in Figure 4. *User-Interface* allows us to specify the target system in C^x programming language. C^x is a superset of C language which permits the user to predefine functions to be mapped in hardware or software. The *translation* unit translates C^x specification into an Extended

Syntax Graph (ES graph) which is kept in a data repository called *Internal Graphs* to be further used by the *Simulator*, *partitioning* mechanism, and the *conversion* unit.

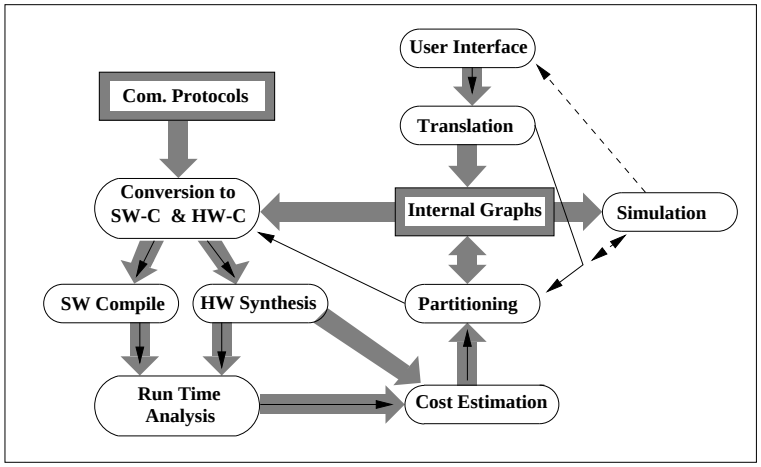

Figure 4: The architecture of the *COSYMA* approach.

3.1 Communication and partitioning mechanism

In *COSYMA*, the partitioning procedure is a kind of optimization problem. The user defines a set of hardware modules such as ALUs, multiplexers, etc., and the partitioning algorithm searches to find the minimum set of software code such that if implemented in hardware, will cause sufficient speedup according to the run time analysis. Time constraints and hardware costs are the main criteria for partitioning. At first all the system functionalities are specified in software. The partitioning procedure tries to move segments of codes from software to hardware to meet the time constraints (with the assumption that hardware is faster than software). The *partitioning* unit gets the segments of codes in ES graph form (from the repository *internal graphs*) and successively moves them from software to hardware. Communication macros are used to maintain the links between parts of the graph moved to hardware with those still are in software. Figure 5 depicts the system as modules and the communication links. The relation of the modules such as procedure calling and parameter passing are shown as edges. Whenever a module is moved to hardware, a set of communication links are established. This causes a communication overhead which should be considered in the cost estimation. In Figure 5 Two modules A and B are already transfered to hardware and the partitioning procedure is now transferring

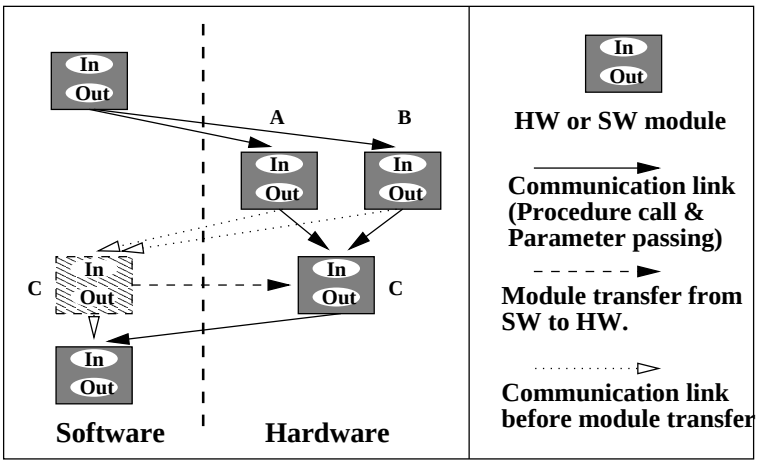

Figure 5: Transferring a module from Software to Hardware.

the module *C* and its links to hardware. The actual communication of the modules in software and hardware is implemented with a very simple protocol, using shared memory as the media.

For each block of codes in software an early estimate of time speed-up is assigned by the user which is a guide for the first partitioning. There is also a user-defined speed up limit which dictates to the system when to stop the partitioning process. For successive partitioning loops, the actual speed-up of the modules will be used instead of the estimated one. The initial module-partitioning is accomplished by the *partitioning* unit, using the estimated speed-up data for the modules. The resultant software and hardware modules are stored back to the *internal graph* repository.

The main partitioning loop starts by an inverse translation of the software and hardware parts form graphs to programming languages. The software modules are translated to *C* programs and the hardware modules are translated to *hardware C*. This conversion generates the specification of the partitioned system in standard languages which allows the use of standard development tools. These operations are represented by the unit *Conversion to SW-C & HW-C*. In the process of generating the source codes, communication macros are also inserted.

The *SW compile* unit compiles the generated *C* program to produce object code for the target processor. The *hardware C* program is a form of hardware description language which is used by the high-level synthesis system *OLYMPUS* [3] from Stanford university. The *OLYMPUS* system generates the hardware functions and implements them as functional units of the co-processors or peripheral units. The whole functionality of *OLYMPUS* is represented by the *HW synthesis* unit. The tar-

get system specified by the *OLYMPUS* system and the software code provided by the *SW compile* unit are fed to the unit *run time analysis*. In the latter unit the interfaces between components are established and a run time analysis is performed to get the timing data. To obtain a criterion for evaluation of the synthesized system, the *Cost estimation* unit uses a cost function which is parameterized by the timing data and hardware cost. Timing data is the execution time of the whole system, and the hardware cost is approximated by the number of synthesized hardware components. The resultant evaluation indicates whether another iteration of partitioning loop is required or not. If the design is not yet optimal, i.e. it can not satisfy the required time speed up assigned by the designer, the *partitioning* unit tries to transfer extra modules to hardware or move some modules back to software. To perform this task, the *partitioning* unit uses actual timing data of the modules, obtained in the run time analysis of the system.

There is also a simulator for ES graphs which allows simulation of a C^x description (with parallel processes). The simulator aids the designer to detect the possible errors in the system specification before any effort for system co-designing, thus preventing extra cost caused by the design of an erroneous system.

4 The CASTLE system

The *CASTLE* system (Co-design And Synthesis TooL Environment) [6] is an automated co-design environment with no pre-defined target architecture. *CASTLE* emphasizes the testability of the target system. Some parts of the system such as partitioning strategies and hardware libraries have recently been completed. System specification is described in one of the programming languages such as *C*, *VHDL*, or *VERILOG*. The architecture of the system is shown in Figure 6. The system is specified in the *user interface* unit, and the specification is kept in a data repository *specification* for system simulation. It is then translated into a series of *internal graphs* in two levels of abstraction. The first layer with higher level of abstraction (*software view*) describes the behavior of the system in the form of control and data flow graphs (CFG/DFG). The second layer, represented by finite state machines, is extracted from CFG and DFG. These FSMs manifest the *hardware view* of the system and have more implementation related details. The partitioning process, represented by the *partitioning* unit, is based on information that exist in *internal graphs*. The graph format allows the process to detect parallel, sequential, and behavior-control parts in order to partition the system. After partitioning the system specification into hardware and software parts, the main task is how we can select the proper components to correctly execute the software (handled by processor), and to perform the hardware operations (implemented in peripherals). The target architecture in the

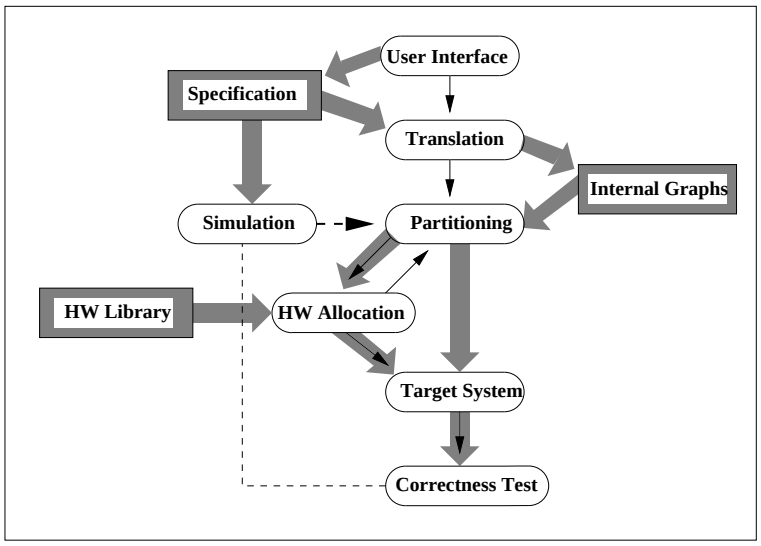

Figure 6: *CASTLE* architecture

CASTLE system is not a pre-defined architecture and can be synthesized in different ways which are not discussed here.

The *HW library* is a collection of available complex components. The *HW allocation* unit is responsible for finding the most proper components, from library, which can perform the operations assigned to the hardware and to select the appropriate processor. The *target system* unit connects the hardware components to the processor and creates a complete system. It also provides the necessary interface among modules. This is a critical task from the *CASTLE* designers' point of view, since the correct operation of each system component is the major goal of the *CASTLE* approach. Therefore, the final operation on the synthesized system is the correctness test which assures the resultant system conforms with the system specification. The *correctness test* unit aims to check all parts of the system to make sure their correct operations. It is responsible to automatically generate test cases to be run on the synthesized system. Some tools and methods exist to test high-level behavior of the system by converting the system behavior to finite state machines. The test based on this technique can not be applied to the low-level structure of the system. Hardware techniques for test case generation, which is applied to the low level structure, are based on functional model of the system. Co-designed systems involve both high level and low level views of a system. Test case generation for the co-design systems requires a combination method for test of both high-level and low-level model of a system. This method is still under investigation. The *simulation* unit simulates those

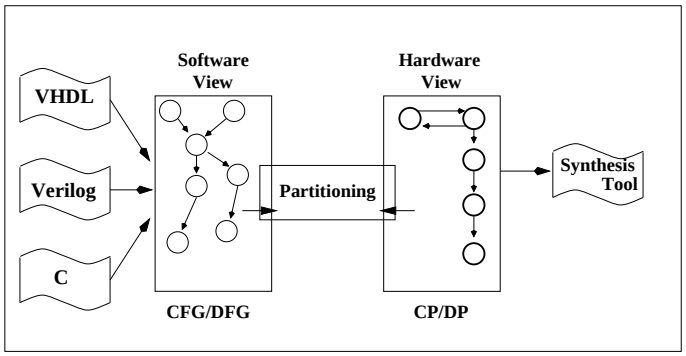

Figure 7: System intermediate representation in *CASTLE* approach.

characteristics of the system which are tested and the results are compared. In case of an unsatisfactory result, the partitioning procedure is resumed.

5 SAAM: Software Architecture Analysis Method

SAAM, a joint development by R. Kazman, L. Bass, G. Abowd, and M. Webb [1], is a method for analyzing of large software architectures. Non-functional qualities of software systems (modifiability, scalability, security, etc.) were used as the evaluation criteria in the SAAM paper. The method had some shortcomings which in the next attempt to improve the method, they changed the focus from non-functional qualities as the base for evaluation, to task-scenarios [2]. In order to apply SAAM to the co-design domain, the system architecture should be an integrated environment, not individual design tools combined by a tool-integration environment such as in MARVEL system [15].

5.1 Overview of the SAAM method

The following descriptions of the SAAM method are excerpts of the paper [2]. Readers who are familiar with the topic may skip this section.

A functional partitioning specifies the distinct functions or services that a system must provide. It is a means for understanding what is common between all applications in a given problem domain. We will use *task scenarios* to develop a functional partitioning. These task scenarios are not intended to cover the complete usage of the system, therefore the resultant functional partitioning may not cover all aspects of the system developed in a particular domain.

In some mature domains such as databases or compilers an agreed on functional partitioning exists. These functional partitioning often come in the form of *reference architectures*. In such cases, this functions may be used instead of generating them from task scenarios. The 8 steps of the SAAM method are summarized below:

1. Define a collection of task scenarios that illustrate the kinds of activities the system must support. These will reflect the non-functional qualities of interest. The scenarios will also present interaction from different roles, such as end user, system administrator, maintainer, and developer.

2. Develop a functional partitioning that manifests the ramifications of the task scenarios. The various functions or services introduced in the partitioning are the result of one or more scenarios. Maintain a coupling between the elements of the functional partitioning and the scenarios that each supports.

3. Classify the task scenarios as *direct* or *indirect* by indicating whether they are executed by the functions and services which are already existed in the architecture (these are direct tasks), or whether the architecture requires some modification in order to include the services (indirect tasks).

4. Express the candidate architecture in a common syntactic architectural notation.

5. Map the functional partitioning onto each candidate architecture.

6. For each direct task scenario, determine whether the target system supports this task (by feature inspection). That is, a direct task can be executed by the functions in the functional partitioning. The evaluation is binary; if the candidate architecture supports the task scenario, it will receive (+) for its evaluation, otherwise will receive (-).

7. Identify the functions associated with the indirect task scenarios. For each function, inspect the structural elements to which it is allocated in the candidate to determine whether any other function is computed within the same structural element. If so, then give the candidate a (-) for the scenarios associated with the function. If not, then give the candidate a (+) for those scenarios.

8. Finally, an overall ranking of the candidate architectures is developed by weighting each scenario and using that weighting to interpret the ratings on the individual scenarios.

6 SAAM method on three co-design approaches

In this section we apply the 8 steps of SAAM method on three HW/SW co-design approaches we already described. These candidate architectures are *UNITY, COSYMA, and CASTLE*, described in sections 2, 3, and 4 respectively.

6.1 Defining the task scenarios

As is described in step 1, by inspection of the three co-design systems we should develop a list of task scenarios. The scenarios are in conjunction with two roles in the co-design environment: *designer* who uses the co-design system to produce a mixed HW/SW target system, and *developer* who modifies parts of the co-design system to install the new functions.

The following task scenarios are related to the role of *designer*:

- **Allocating parts of the specification to hardware or software:** In this scenario, the designer forces the design tool to allocate parts of the specification into software or hardware, hence overruling the automated partitioning mechanism for those parts. This scenario assists the designer to start the partitioning algorithm with specific components or to test an implementation alternative. It manifests the *flexibility* (also called *migratability*) of the co-design system.

- **Changing the hardware components:** In this scenario the designer modifies the attribute list of the available components in the hardware library, or includes (excludes) hardware components. This scenario in fact changes the target system which executes the software part and reflects the *modifiability* quality of the co-design approach.

- **Adding pre-existing HW or SW specifications to the target system:** The aim of this task scenario is to equip the co-design system with a set of efficient and reliable software and hardware modules which are already tested and used. In order to use these libraries of module specifications, the system should use the standard tools for software compilation or hardware synthesis. This *re-usability* of elements is a key characteristic of the new development environments.

- **Trying several implementation alternatives for the target system:** Since it is difficult to come up with the best partitioning result on the first try, a mechanism is required to automatically repeat parts of the (or the whole) design process to achieve a design alternative with optimal performance. In this process the mechanism should use some evaluation criteria defined by the designer.

- **Change of the HW/SW communication protocol:** The communication between modules in hardware and software is an important factor in the design process. It describes how and when the modules should send or receive data. Communication is important in the sense that it directly affects the time factor in the cost function. The designer may desire to test different methods of communication and observe the effects on the overall system performance. This reflects the *modifiability* characteristic of the design system.

- **Simulating the system specification:** The design process is started by defining the system specification. Producing an error free specification can save human effort and money. Using a simulation utility to test the specification and assurance of its correct behavior is the best way of producing a bug-free specification.

The following task scenarios are related to the role of *system developer*. The scenarios are in conjunction with the provision of modern design facilities for this domain.

- **Interfacing to a high level synthesis tool:** This task scenario investigates the possibility of replacing the conventional editor, as the user-interface, with one of the state of the art graphical interfaces. These graphical tools translate the system specification into one of the common textual languages. This modification provides us with the use a high level synthesis facility in the co-design domain. This improvement represents the *Integrability* of the system.

- **Adding a target-system demonstration utility:** Creating a correct system is the goal of the system development tools. Each co-design system has a certain method of comparing the target system against its specification. This scenario suggests adding a general simulation tool which assists the user in interactively testing the performance of the final system. It draws the component layout of the system and permits the user to provide input data and observe the outputs of the system. It also presents some statistics of the system as a benchmark. This scenario is an indication of *Integrability* of the co-design method and serves for checking the *correctness* of the target system against its specification.

6.2 Functional partitioning

In accordance with step 2 of the SAAM method, we now extract a series of functions and services from the task scenarios we presented in the previous section.

Considering the task scenario:

- *Allocating parts of the specification in hardware or software.*

we need a user-interface and a feature which is supported by the HW/SW partitioning mechanism to allow the user to allocate parts of system specifications in hardware or software. In most cases the initial partitioning is determined by the designer which requires the following services. Acronyms are used to refer to the functions.

UI: User Interface.

SU: Support for User defined partitioned elements.

For the scenarios

- *Changing the hardware components.*
- *Adding pre-existing hardware or software specifications to the system.*

we require a hardware library to accommodate the components, a hardware allocation function which assigns different hardware specifications to available hardware components, and a software allocation function which translates (compiles) the software specifications to the machine language of the target processor.

LB: hardware LiBrary.

HA: Hardware Allocator.

SA: Software Allocator.

Considering the following task scenario,

- *Trying several implementation alternatives for the target system.*

we need to somehow simulate the target system in order to acquire data from its dynamic behavior (run time analysis), and its hardware or software cost. This simulation is different from the simulation task which is usually performed for testing of the system specification. We need to generate cost functions according to the information provided by the target system simulation, and finally the cost functions should be compared against the user-defined constraints. Therefore the following functions are required.

SM: target system SiMulator.

CF: Cost Function generator.

EV: EValuator.

For the task scenario:

- *Adding a target-system demonstration utility.*

similar to the previous scenario, we first need to connect different parts of the target system and simulate its functionality, then using a demonstration utility we can see the system layout and provide some input data to it and observe the output of the system. Therefore, we need the following functions

SM*: [4] target system SiMulator.

DU: Demonstration Utility.

Regarding the task scenario:

- *Change of the HW/SW communication protocol.*

we need a protocol-generator function which produces different communication protocols. A designer, using a user-interface, defines the protocol and the protocol-generator defines and schedules the communication sequence among the modules. Therefore we need:

PG: communication Protocol Generator

UI*: User Interface.

The next task scenario,

- *Simulating the system specification.*

needs a simulator tool to simulate the system specification. The simulator aids the designer to check the design functionality for error detecting.

SIM: specification SIMulator.

SP: SPecification repository.

In the following task scenario,

- *Interfacing to a high level synthesis tool.*

[4] * The function is already defined.

two approaches are imaginable. Both use graphical editors and other facilities of the tool to specify the design. In the first approach the code generation engine of the high level synthesis tool is not used and the tool directly produces the internal graphs of the co-design system (e.g. CFG/DFG in *CASTLE*, or internal graphs in *COSYMA*). In the second approach the designer uses the code generator of the specification tool and produces the source code in one of the common programming languages (i.e. *C*, or *VHDL*). The produced code is in fact the system specification which is then used by the partitioning mechanism. For this analysis we require a graphical editor to produce the specification of the system in *Statecharts* or *state transition diagrams*, a graph translator to produce the internal form of specifications in each co-design approach, and a code generator to produce the target codes.

GE: Graphical Editor.

GT: Graph Translator (for the first method).

CG: Code Generator (for the second method).

After the functional partitioning process, step 3 of the SAAM method suggests to classify the task scenarios as *direct* or *indirect*. Table 1 shows the task classifications.

As we mentioned earlier, when there are several systems to be compared, it is helpful to draw their architectures in a common notion. This is the step 4 of the SAAM method. We already presented our candidate architectures in a simple common notation. These architectures are repeated in the next section.

In order to apply the remaining steps of SAAM, we separately discuss each of our three candidate architectures.

6.3 Candidate architecture *UNITY*

The *UNITY* system was described earlier in section 6.3. In pursuing step 5 of SAAM, those functions which exist in *UNITY* system, are indicated in Figure 8. The architecture of the system has been annotated with the functional units introduced by indirect task scenarios (dashed boxes). Now, we investigate whether the candidate system supports the direct tasks, or not (step 6). By feature inspection of the *UNITY* approach and functional allocation of the direct task scenarios to the modules of the architecture, we are convinced that this system supports all direct scenarios except one. It can not simulate the system specification so it gets a (-) for this part.

The following section investigates the effect of indirect tasks to the system.

- **Change of the HW/SW communication protocol:** For this task scenario, we need a protocol generator (unit PG in Figure 8) which also interacts with

	Task Scenario	Classif-ication	Functions	Quality factor
1	*Allocating parts of the specification in hardware or software*	*direct*	UI, SU	*Flexibility*
2	*Changing the hardware components*	*direct*	LB, HA	*Modifiability*
3	*Adding pre-existing hardware or software specifications to the system*	*direct*	LB, HA, SA	*Re-usability*
4	*Trying several implementation alternatives for the target system*	*direct*	SM, CF, EV	*Efficiency*
5	*Simulating the system specification*	*direct*	SIM, SP	*Testability*
6	*Change of the HW/SW communication protocol*	**indirect**	**PG, UI**	*Modifiability*
7	*Adding a target-system demonstration utility*	**indirect**	**SM, DU**	*Integrability*
8	*Interfacing to a high level synthesis tool*	**indirect**	**GE, CG**	*Integrability*

Table 1: Functional decomposition of the task scenarios, and their classification into *direct* and *indirect*.

the user. The communication protocol is extracted from the *interface graphs* which is created in unit *alloc/sched* (Refer to section 2). Unit PG is attached to unit *alloc/shed* to manipulate the *interface graphs*. In order to change the protocol, the interface graph should be changed. This change is not localized to one location and propagates through the system. The *simul/eval* should be modified to use different protocols for simulations. Therefore *UNITY* system gets a (-) for this task.

- **Demonstration of the target system:** In this scenario, unit *simul/eval* provides the information of the simulated target. This unit can then communicate with the demonstration utility (unit DU in Fig.) to show the target system behavior. The user-inputs, through unit DU, can be passed to the target-simulator unit and after being processed, the results are sent back and shown by the demonstration screen. This modification does not affect other parts of the system, therefore get a (+).

- **Interfacing to a high level synthesis tool:** The *UNITY* language, though

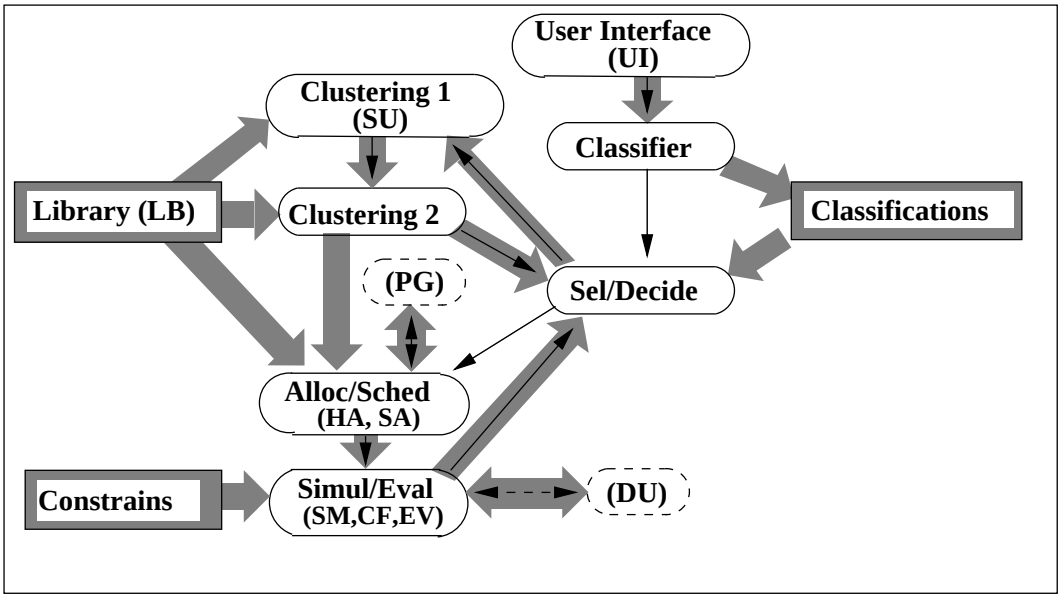

Figure 8: *UNITY* System with function allocations and changes to its architecture.

it is good for system specification, has its own peculiarities and is not supported by high level synthesis tools. The first approach for implementing this task scenario is to generate the parse tree, used by the *classifier unit*, from the graphical specification methods. In doing this, the classifier task should be changed to extract the implementation alternatives for the *UNITY* elements from the graphical tool. The second approach uses the generated code from the graphical tool as the *UNITY* system specification. Existing graphical specification tools do not generate the *UNITY* language. None of these methods gives us a promise for successful implementation of this task scenario, therefore the *UNITY* architecture receives a (-) score for it.

6.4 Candidate architecture *COSYMA*

The *COSYMA* approach is presented in section 3 and its architecture is repeated in Figure 9. For this system also, we go through steps 5, 6 , and 7 of the SAAM method. Figure 9 shows the mapping of the partitioned functions to the structure of the system (step 5). Feature inspection of this system with respect to the direct task scenarios (step 6) indicates that the *COSYMA* system supports all the direct tasks and therefore gets a (+) for that.

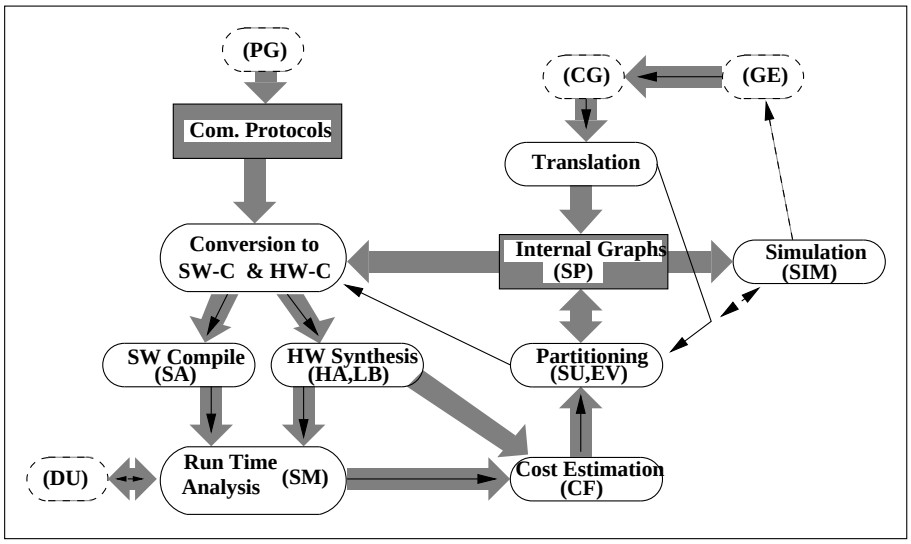

Figure 9: *COSYMA* System with function allocations and changes to its architecture.

The investigation of three indirect task scenarios are described below:

- **Change of the HW/SW communication protocol:** This scenario needs the installation of a protocol generator (PG). Fortunately the system has already isolated the communication protocols in the repository *com. protocols*. Therefore the protocol generator can easily access and change the protocols without any interference with other modules functionalities. The architecture gets a (+) for this scenario.

- **Demonstration of the target system:** This task scenario is also easy to implement. The hardware elements, i.e. co-processors, and object code for a target processor, are fed to the unit *run time analysis* where the target system is simulated. This unit can communicate the system simulation data with the module *demonstration unit*. The *COSYMA* system gets another (+) for this feature.

- **Interfacing to a high level synthesis tool:** To investigate the applicability of this scenario, we consider the specification language for this approach, C^x, which is a superset of C language. C^x contains multitasking and process communication features and its skeleton is C. In order to use a high level synthesis tool with this system, the code generator function (CG) of the tool should be modified to be matched with the extra features of the C^x language over

conventional *C*. This adjustment may be difficult or not, depending on the different tools. In fact high level synthesis tools are capable of designing parallel processes and obviously this parallelism is reflected on their generated output codes. As a result the interface of a high level synthesis tool to the *COSYMA* system is achieved by applying the output of the code generator tool to the *translation* unit of the architecture and designing the system with the graphical editor (GE). *COSYMA* gets a (+) for this scenario.

6.5 Candidate architecture *CASTLE*

The co-design system *CASTLE* was described in section 4. We apply step 5 of SAAM and map the decomposed functions into the structural elements. The architecture is shown in Figure 10. Feature inspection for direct task scenarios in Table 1 indicates

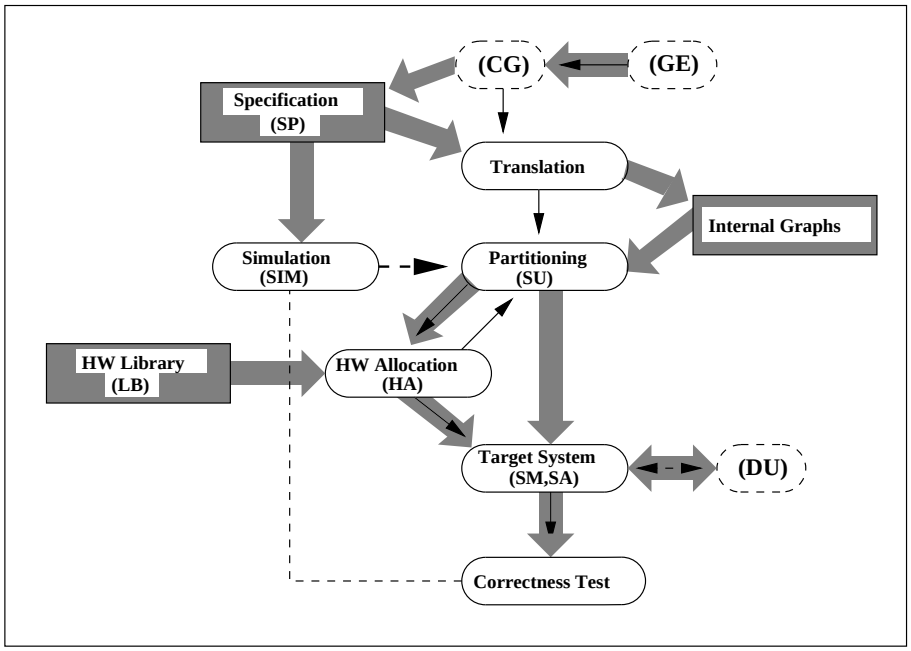

Figure 10: *CASTLE* System with function allocations and changes to its architecture.

that the *CASTLE* system can handle all direct tasks except one. Since the partitioning mechanism of the *CASTLE* system is not published, we are not sure about the proper support of the task scenario, *Trying several implementation alternatives for the target system*, in this approach. Although the target system correctness is a criterion for repeating the partitioning process in the *CASTLE* system, still other

important parameters such as area, timing, etc., defined by the designer and illustrated in the cost functions, are not used as the criteria for resuming the partitioning process. This is implied from the existing system architecture which lacks a structural block assigned for the target system evaluation against these parameters. *CASTLE* receives a (-) score for this deficiency. In accordance with step 7, three indirect tasks scenarios are investigated in the following part.

- **Change of the HW/SW communication protocol:** Since the partitioning process of the *CASTLE* is recently developed, there is not sufficient information about the communication method among the modules in hardware and software, so we can not judge the merit degree of this task scenario.
- **Adding a target-system demonstration utility:** It seems that the implementation of this scenario is as simple as in the *COSYMA* system, since the *target system* unit has the simulation information of the target system and therefore the demonstration utility can be interfaced to this unit. This task has no interference with other tasks of the system, therefore it scores a (+) for the system.
- **Interfacing to a high level synthesis tool:** The system specification in *CASTLE* is defined in common languages such as *C*, *VHDL*, or *Verilog*. This means that a high level synthesis can directly be used and the output of the code generation tool can be used as the system specification. Another method described for using a high level synthesis tool is also imaginable for this system. The *internal graphs* of the *CASTLE* approach are in the form of control flow and data flow graphs which can be generated directly by graphical editors of some CASE tools (e.g. *Statemate*). This solution, though it seems ideal, has many consequences. It requires vast changes to some system parts such as *partitioning* and *simulation* unit, which is not desired. Using the first method, this system gets a (+) for the task scenario.

7 Comparisons

Table 2 summarizes the results of the above analysis. In order to evaluate the candidate co-design systems according to the comparison table, the evaluator should assign weightings for task scenarios. Hence, using SAAM as a concrete method for analyzing the system architectures, the evaluator's interpretation regarding the relative importance of the task scenarios is a determining factor. This characteristic provides the SAAM method with a degree of freedom which is helpful for evaluation of the candidate architectures from different points of view. As we mentioned in the

	Supporting direct task scenarios	(indirect) Change of the communication protocols	(indirect) Adding system demonstration utility	(indirect) Interfacing to a high level synthesis tool
UNITY	*All, except System simulation*	–	+	–
COSYMA	*All*	+	+	+
CASTLE	*All, except Trying several imple. alternatives*	*Communication method is not identified yet*	+	+

Table 2: Comparison of the analysis

introduction, the main concern of this paper is the study of some co-design architectures regarding their potential to be equipped with modern design facilities, therefore the indirect task scenarios were created accordingly. With the above objective, we assign the highest weight to the indirect task, *interfacing to a high level synthesis tool*, and the next weight is given to the scenario *Adding system demonstration utility*. The least important indirect task in this strategy, is the *change of communication protocol* which in different situation may acquire the highest weighting.

Referring to the Table 2, we observe that *COSYMA* has gained all of the scores and therefore it is considered to have the best architecture among the others. The *UNITY* architecture is not suitable for a change of the communication protocols; and regarding the enhancements with modern development tools, it is neither appropriate for interfacing to a high level synthesis tool, nor supports the simulation of system specification. As we know, these shortcomings are important in our discussion, therefore *UNITY* architecture is evaluated as the weakest architecture in this regard. *CASTLE* needs some improvements in its partitioning loop such that it can evaluate the target system against the user-defined constraints. One last point is to be mentioned; the above evaluation does not imply that *UNITY* is not a good co-design system, and in fact it has a sophisticated partitioning technique. What we evaluated here was: how well different co-design architectures could adopt extra features for the future changes or improvements.

8 Lessons learned

In the course of this evaluation we learned the following lessons: (i) the correctness of the SAAM method is directly affected by the selection of the appropriate scenarios. To analyze a system correctly, a domain expert and information providers (e.g., end-user, developer, maintainer, etc.) are needed to set up a proper set of task scenarios which reflect the various uses of the systems in the field. Otherwise the architectures may be evaluated with incidental scenarios, producing wrong results. (ii) It was not easy to produce homogeneous architectural diagrams of different systems, with the same notations and level of details. This was partly because each designer tried to elaborate particular aspects of the system, and partly because our required information was not directly presented in the resources we used. The required functions by task scenarios implicitly dictated us the right level of abstraction in the representation of diagrams.

(iii) The evaluation of these three HW/SW co-design systems brought us a valuable knowledge of this field, which could not be acquired with isolated study of each system. Although, SAAM is not intended for designing a new system, its notion of task scenarios and their break-down into a set of functions are helpful to the structural design of a HW/SW co-design system. The functional decomposition of the direct task scenarios provides us with the common functional qualities we expect from the new system. Different allocations of the functions to the structural blocks of the system can drastically affect the quality of the system architecture. We expect a good architecture to show low coupling and high cohesion in terms of the functional allocations to the structural elements of the system. To provide low coupling, we do not distribute the functions related to a single scenario among a large number of structural components. To provide high cohesion, we try to prevent the allocation of the functions related to different scenarios into single structural element. The above design considerations are the implications of the evaluation criteria used in the SAAM method, which assists us in designing a new system.

9 Conclusion

In this paper we have introduced the subject of hardware/software co-design and three competitive approaches in this field were briefly presented. We tried to establish a link between the concerns and facilities provided in software system engineering with this new research area. Illustrating this connection, a software system architecture-based analysis method (SAAM) was used to analyze and compare our selected approaches in co-design. In this way we succeeded to show the strengths and weaknesses of the candidate architectures regarding their capabilities to accept modern design facilities available in related disciplines (software or hardware).

References

[1] R. Kazman, L. Bass, G. Abowd, M. Webb.*SAAM: A Method for Analyzing the Properties Software Architectures.* Proceedings of ICSE 16, May 1994, pages 81-90.

[2] R. Kazman, L. Bass, G. Abowd, P. Clements. *An Architectural Analysis Case Study: Internet Information Systems.* Presented to ICSE 17 Workshop on Software Architecture, 1995.

[3] G.D. Micheli, D.C. Ku, F. Mailhot. *The OLYMPUS synthesis System for Digital Design.* Design & Test Magazine, pages 37-53. October 1990.

[4] A. Geist, A. Beguelin, J. Donggarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3.0 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Tennessee, February 1993.

[5] M. Cagan. *The HP SoftBench Environment: An Architecture for a New Generation of Software Tools.* Hewlett-Packard Journal, June 1990, pages 36-47.

[6] U. Steinhausen, R. Camposano, H. Gunther. *System-Synthesis Using Hardware/Software Co-design.* International Workshop on Hardware-Software Co-Design, Cambridge, Massachusetts, October 1993.

[7] J. Henkel, Th. Benner, R. Ernst. *Hardware generation and partitioning effects in the COSYMA system.* International Workshop on Hardware-Software Co-Design, Cambridge, Massachusetts, October 1993.

[8] E. Barros, W. Rosenstiel, X. Xiong. *Hardware/Software Partitioning with UNITY.* International Workshop on Hardware-Software Co-Design, Cambridge, Massachusetts, October 1993.

[9] S. Antoniazzi, A. Balboni, W. Fornaciari, D, Sciuto. *A Conceptual-Level Approach to Embedded System Design.* International Workshop on Hardware-Software Co-Design, Cambridge, Massachusetts, October 1993.

[10] S. Kumar, J. H. Aylor, B. W. Johnson, W. A. Wulf. *Exploring Hardware/Software Abstractions & Alternatives for Co-Design.* International Workshop on Hardware-Software Co-Design, Cambridge, Massachusetts, October 1993.

[11] V. Carchiolo, M. Malgeri. *Behavioral Approach to System Co-Design.* International Workshop on Hardware-Software Co-Design, Cambridge, Massachusetts, October 1993.

[12] K. Olukotun, R. Helaihel. *Automating Architectural Exploration with a Fast Simulator.* International Workshop on Hardware-Software Co-Design, Cambridge, Massachusetts, October 1993.

[13] K. OBrien, T. B. Ismail, A. A. Jerraya. *A Flexible Communication Modeling Paradigm for System-level Synthesis.* International Workshop on Hardware-Software Co-Design, Cambridge, Massachusetts, October 1993.

[14] S. Lee, J. M. Rabaey. *A Hardware-Software Co-Simulation Environment.* International Workshop on Hardware-Software Co-Design, Cambridge, Massachusetts, October 1993.

[15] B. Dinler, B. Kramer. *Integrating a CAD Tool Box with a Software Process Environment: A Case Study.* International Workshop on Hardware-Software Co-Design, Cambridge, Massachusetts, October 1993.

[16] A. Juntunen, J. Kivelae, A. Reinikka. *Real Time Structured Analysis in System level Design of Embedded ASIC's.* Microprogramming and Microprocessing, volume 24, page 449-454, North Holland, 1988.

[17] R. Gupta, G. De Micheli. *System level Synthesis Using Re-programmable components.* In proceedings of EDAC, pages 2-7, Brussels, Belgium, March 1992.

[18] E.D. Lagnese. *Architectural Partitioning for System Level Design of Integrated Circuits.* PhD thesis, Carnegie Mellon University, March 1989.

[19] F. Vahid, D.D. Gajski. *Specification Partitioning for System Design.* In proceedings of the 29th Design Automation Conference, pages 219-224. Anaheim-California, 1992.

[20] M.C. McFarland, A.C. Parker, R. Camposano. *The high-level synthesis of digital systems.* Proceedings of the IEEE, 78(2): pages 301-318, February 1990.

[21] R. Camposano, R.K. Brayton. *Partitioning before logic synthesis.* In proceedings of the IC-CAD'87, pages 237-246. Santa Clara, Ca, November 1987.

[22] G. Boriello, K. Buchenrieder, R. Camposano, E.A. Lee, R. Waxman, W.H. Wolf. *A d & t roundtable hardware/software co-design.* IEEE design & test of computers, pages 83-91, March 1993.