

Towards a Theory for Hardware/Software Codesign

Markus Voss

Universität Karlsruhe, IMA
Karlsruhe, Germany
e-mail: mvoss@ira.uka.de

Tarek Ben Ismail

INPG, TIMA
Grenoble, France
e-mail: ismail@imag.fr

Ahmed A. Jerraya

INPG, TIMA
Grenoble, France
e-mail: jerraya@imag.fr

Karl-Heinz Kapp

Universität Karlsruhe, IMA
Karlsruhe, Germany
e-mail: kapp@ira.uka.de

Abstract

This paper aims at a theory for hardware/software codesign. We approach this goal by investigating system design according to the allocation principle which is a systems engineering approach to mixed hardware/software systems design. The associated process steps are system-level design including partitioning and communication synthesis by channel mapping and binding. A strengthened system-level design and synthesis process makes codesign with delayed technology dependent partitioning possible. In addition we investigate possibilities for automating these processes by tool support.

Keywords: System-Level Design, Hardware/Software Codesign, Design Process, Design Theory

1. Introduction

As computer-based systems become more and more complex the need for methodological support in managing the development process and life-cycle of these systems becomes more and more urgent. This need is approached by both the definition of systems and software development standards and by a huge amount of methods and tools developed to aid in either the operational tasks of systems development or to support managing development and maintenance processes. It is a symptom of process maturity if the underlying system theory is understood and agreed upon by the community, reflected within the standards, and supported by the methods and tools. In other words one can conscientiously talk of engineering if things fit together that way.

It is obvious however, that while the process of building software systems (to be installed and executed on standard hardware components) is quite mature in that sense, the process of developing mixed hardware/software systems in practice is still dominated by a great degree of

heuristics. This observation leads us to conclude that there is a lack of basic system theory for system-level design and how to systematically proceed towards a detailed design of mixed hardware/software solutions, in other words hardware/software codesign.

Within that paper we try to make a contribution towards defining that kind of theory. We will define a terminology for system-level design work, outline the basic engineering steps in hardware/software codesign with communication synthesis being the point of major concern, and sketch possibilities for tool support. In addition we will map the theoretic knowledge into a generic framework for concrete mixed hardware/software systems engineering process standard definition.

2. Design Theory

2.1 Services

System design is the activity of mapping a conceptual model's (model of the system's behaviour abstracting from implementation details used as a media for requirements analysis) information into a physical structure of cooperating components in order to define an architecture that satisfies the users needs in terms of both functional and non functional requirements. The system is assembled together from physical components of defined functionality. This work covers both the system's architectural definition as well as the detailed, internal

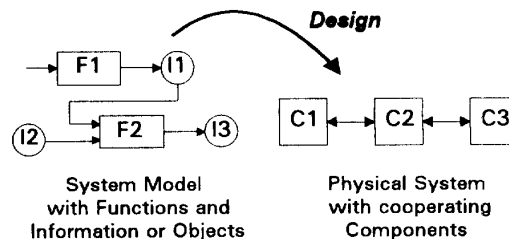


Figure 1: System Design

design of the components constituting this architecture. Figure 1 illustrates this general meaning of design.

Highly valuable for systemizing design work is the term *service*. A service is a physical component of a system, that can be ordered from outside, this is from other system components, to carry out some sort of functionality to assure the performance of some part of the system's overall functionality. A component of that kind consists of *hardware and/or software* parts. Software parts of a service are linked data structures and algorithms (*modules*) which assure wanted functionality from the software engineering point of view. Hardware parts of a service are both components directly assuring the needed functionality or components the software parts can be installed and executed upon.

Especially in automation where temporal and performance requirements tend to be very important the system designer should be given the highest degree of freedom to decide upon which technological solution is the most suitable for his problem. Services may be dedicated hardware components (ASICs, microcode/datapath), intelligent subsystems of mixed hardware, software and control parts or software for standard components and operating systems.

Once specified in terms of its functionality a service may manifest itself in a *variety of products* of different designs, each with its own specific non functional quality measures. Design itself then has the character of choice between different alternatives which is a prerequisite to satisfy design's inherent nature of being a matter of trade-off decisions. The service is the entity of managerial consideration within the design process both in terms of having non functional quality measures and re-use properties assigned. For more details on the nature of service-oriented design we refer to [1] or [2].

2.2 Allocation

The design mapping from above can be partitioned into several steps or distinct tasks. The first way of partitioning design work is in terms of a design's *location dependent and location independent* characteristics. This is done by separating issues concerned with 'logical' unit building like process structure definition or decisions in terms of parallelism from issues concerned with physical structure building like choosing sorts of physical components and interconnection media. These 'logical' units will from now on be referred to as *Design Units* (DU). They constitute a *location independent system design*.

Building a location independent system design is very often called *system-level design*. All sorts of languages are proposed for performing that task (SDL, StateCharts, Lotus, Estelle, OCCAM, CSP-like languages etc.). These have in common that one models the system as a (possibly hierarchically ordered) set of parallel and interacting extended finite state machines (EFSM). The means of communication may be abstracted as happening via channels and the models communication protocol assumptions is where the approaches differ the most. That observation calls for a sort of intermediate form for system-level design description general enough to model all forms of interaction from the established notations. We return to that issue when talking about communication.

One may regard executing both tasks mentioned above as building two distinct sorts of system structures. Resulting from separating these issues is a third step of integrating these two structures again for the system structure of cooperating services by mapping design units onto physical components. Physical components therefore can be regarded as *places* whereon to *allocate* defined

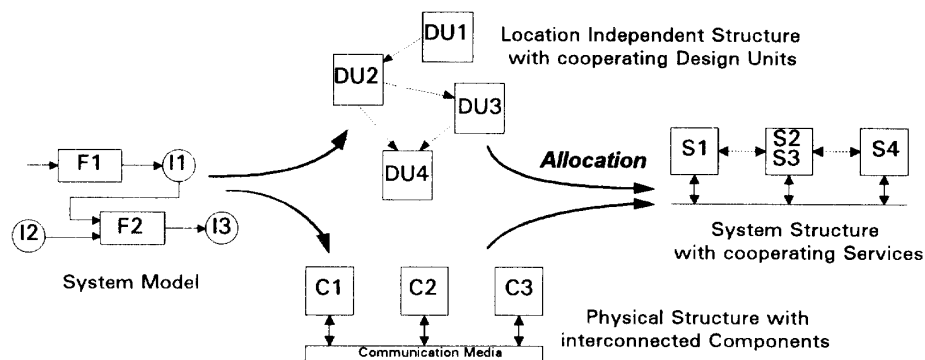


Figure 2: System Design by Allocation

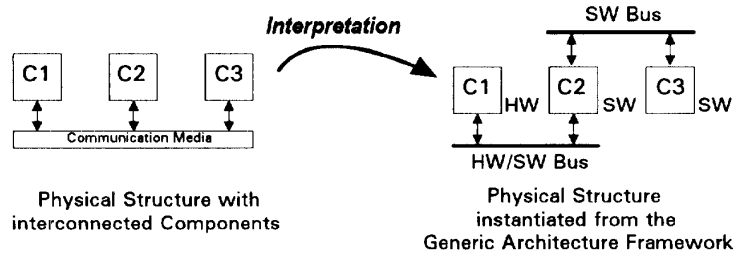


Figure 3: Interpretation of the Physical Structure

functionality in terms of design units. We call this process system design according to the *allocation principle*. It is illustrated in figure 2.

In software engineering this separation corresponds to a separation between hardware and software issues. Design units correspond to software modules and physical components correspond to the hardware the software can be installed and executed upon.

In mixed hardware/software systems engineering the physical components may be both components directly assuring the needed functionality or components which are hosts for software. On a sufficient level of detail these are *either one or the other*. If this level of detail is matched, the physical structure of interconnected components not only covers what may be called the architecture interconnection structure, as e.g. described in architecture interconnect diagrams from RT/SD [4], which is a system-level information, but also hardware/software partitioning decisions.

These decisions consequently also apply for the location independent structure of design units. On a sufficient level of detail (in terms of the hierarchy of design units stated above) this structure not only covers what may be called the architecture flow structure (described in architecture flow diagrams from RT/SD), which also is a system-level information, but the same hardware/software partitioning decisions addressed above.

Note, that going from system-level issues down to

hardware/software design issues following these ideas is only a matter of abstraction or *level of detail*. The conclusion is, that using the introduced abstractions, one can seamlessly proceed from the system-level view to the hardware/software view.

2.3 Generic Architecture Framework

We first concentrate on the physical structure of interconnected components and investigate what characteristics there are that one can utilize when working towards system synthesis.

Having the either/or classification of physical components from above yields a taxonomy for physical components and interconnection media which can be captured in a *Generic Architecture Framework* (pre-defined general architecture). Components are of *type* hardware or software (software hosts). Interconnection media are of *type* hardware to hardware (HW bus), software to software (SW bus), or hardware to software (HW/SW bus). Figure 3 illustrates the interpretation of the physical structure of interconnected components as an instantiation of the generic architecture framework. After looking at communication, this definition will have to be extended slightly.

There may be many implementations of these generic entities. Software type components are Microprocessors with memory or other programmable components. Hardware type components are ASICs, PLDs, FPGAs, or other existing circuits. Interconnection media are wires or buses of different specification connecting sub-systems. Their mapping into the taxonomy allows for an organization in terms of re-use.

2.4 Service Access

When working on the software side one can easily abstract from the underlying hardware and communication implementation details. Service modules may be regarded as only having access to other services

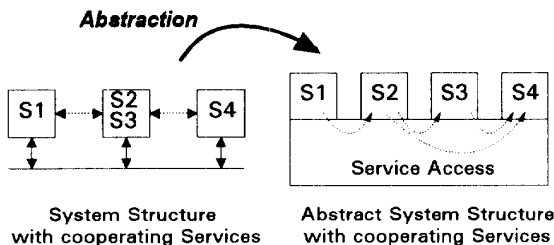


Figure 4: Abstraction from Hardware and Communication

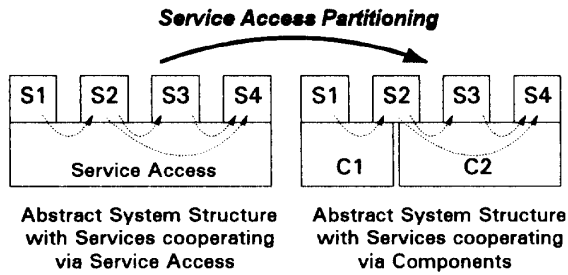


Figure 5: Service Access Partitioning

via a transparent *service access* using functionality provided by a *service access interface*. That service access is the media for all communication among services and realizes conceptual communication protocols to be used by the service modules. This abstraction principle is illustrated in figure 4.

The service access can be regarded as a kind of 'logical system bus' into which services can be 'plugged in' for system integration or configuration purposes. That abstraction effectively separates the design issues as described above and allows for a most natural understanding of the integration or configuration aspects. An implementation of the service access as a basis for software system development can be realized by an extended operating system to be present on all physical components and being accessible (in whatever syntactic form) from within the programming language [5].

In mixed hardware/software design we naturally have to care about the communication details. *Communication synthesis* is an integral part of mixed hardware/software systems design. Due to service-orientation however, one can utilize the same abstraction from above concerned with software system design to form a sound theoretical basis for communication synthesis. Services cooperate by accessing the service access which serves as some sort of co-processor responsible for inter-service communication execution. It realizes the channels the design units use for communication as well as the actual physical links.

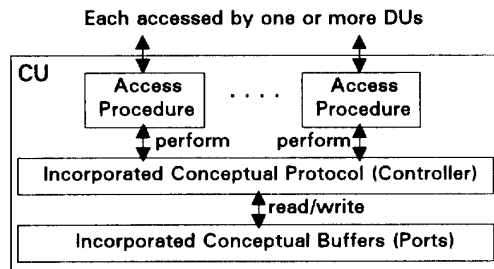


Figure 6: Channel Unit Structure

Because of the inherent heterogeneity of mixed hardware/software systems however, we regard the service access as decomposable into *components* according to *sets of the single channels* of communication between design units. That fact is illustrated in figure 5. Each component hides a special *sort* of communication implementation. The classification may be in terms of protocol specification as well as communication media. With the example from above, C2 may e.g. be the mentioned extended operating system plus underlying communication media. The sets of channels corresponding to the components of the service access are called *Channel Units* (CU).

Channel units may be regarded as consisting of

- access procedures, to be accessed by the DUs
- buffers or ports according to the encapsulated conceptual channels, and
- a controller to perform the access on the buffers according to the protocols.

This is illustrated in figure 6. The process of organizing the communication within the location independent design according to channel units is called *channel unit transformation*. It is illustrated in figure 7.

Note, that channel unit transformation is one way of mapping design descriptions given in one of the established forms mentioned above into an intermediate form, because it allows for more than one form of communication protocol to exist within the same framework. In addition, channel unit transformation is the first step in communication synthesis, namely going from communication specification to communication design.

Before we go into more detail on the channel units internal structure, a classification scheme for the communication protocols, and the process steps in communication synthesis, we will first have a closer look at the location independent structure of cooperating design units regarding system synthesis.

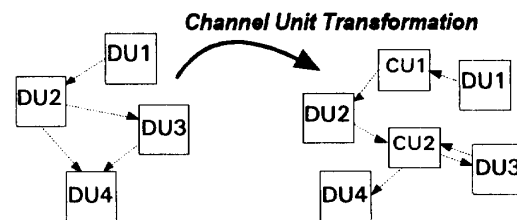


Figure 7: Channel Unit Transformation

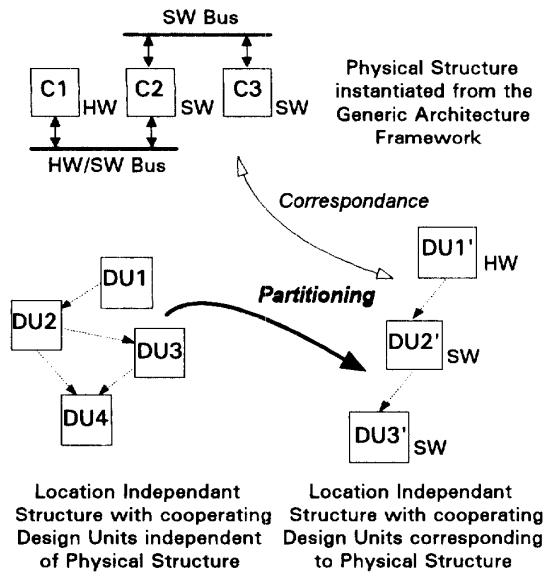


Figure 8: System-Level Partitioning

2.5 Partitioning

Channel units also provide the mechanism for transforming a system-level description completely independent from physical structure information into a system-level description which is more synthesis oriented. In the later case the structure given by design units and channel units is build according to the proposed physical structure of interconnected components. Design units are built matching the physical components and channel units matching the interconnection media.

This aspect of transformation yields a partitioning of the engineering step of location independent structure building into two sorts of activities. One is building that structure regardless of the proposed physical structure and the other is transforming it to better suit synthesis requirements. In [6] the later process is called *system-level partitioning*. It is illustrated in figure 8. As a result the transformed structure matches the level of detail mentioned above where hardware/software partitioning issues are incorporated into the location independent structure of cooperating design units.

Starting from a given system-level design the transformation is done by reordering the hierarchy, merging concurrent processes together to form one process for a better resource utilization, splitting up one process to form concurrent processes for distribution purposes, etc. For more details on transformation

operations and the impacts on the structure of parallel processes we refer to [7].

2.6 Communication Synthesis

By now we have investigated how the two structures build in system design according to the allocation principle are to be detailed or transformed according to system synthesis requirements. What we still lack is a systematic procedure for mapping these onto one another. The major question naturally is what the channel units map to or in other words how communication details are managed and kept consistent going from specification to implementation. This is what we will analyze now.

2.6.1 Protocol Specifications

First one must have a *classification* of conceptual communication protocols as incorporated within the models underlying the different system-level design descriptions. A communication between two design units or processes is primarily characterized by whether reading and writing information is destructive or non destructive (or potentially blocking or non blocking respectively). An orthogonal classification can be made in terms of assumed conceptual buffer size and directionality (unidirectional or bi-directional) with respect to that conceptual buffer.

As an example, the mechanism underlying SDL specifications is non destructive writing, destructive reading, unlimited buffer, and unidirectional flow. The Rendezvous mechanism underlying OCCAM or CSP-like languages is non destructive writing, destructive reading, buffer size one, and unidirectional flow and the one underlying the remote procedure call (RPC) is like the former only with bi-directional flow.

That classification scheme covers all relevant aspects of the protocols and has to be maintained over design and implementation. It is the key to channel unit mapping consistency and channel implementation re-use.

2.6.2 Channel Binding

Having these classifications the first step in communication synthesis is what is called *channel binding* in [6]. Channel binding is the engineering process view on what was called channel unit transformation. It means picking an appropriate channel unit (from a library of channel units) and replacing communication operations due to conceptual protocol specification by accessing the channel unit. This step may

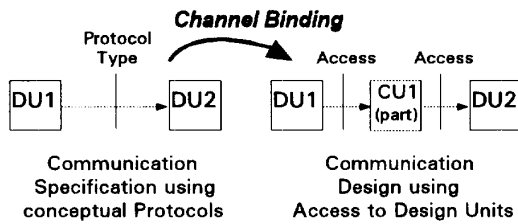


Figure 9: Channel Binding

be regarded as going from specification to design and is illustrated in figure 9.

It is important to notice, that the access itself is also performed according to some protocol. In SOLAR a RPC protocol is proposed. Its characteristics are described above. Regarding the protocol as conceptual in the sense defined above gives the channel units the *character* of additional design units responsible for executing communication between some of the 'primary' design units. Being EFSMs, they hide the communication buffers within their local variables and the control mechanisms within their process specification.

This abstraction corresponds well with the view taken in service access partitioning. The system structure of cooperating services consists of services corresponding to the 'primary' design units and of the components of the service access, which therefore have service character themselves. One may talk of application and communication services.

2.6.3 Channel Mapping

The second step in communication synthesis is called *channel mapping* in [6]. It may be regarded as going from communication design to implementation. Channel mapping is about distributing the channel units functionality (by distributing access procedures, controller functionality and buffers) across units matching physical

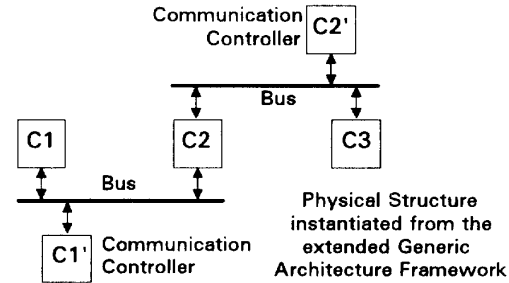


Figure 10: Generic Architecture Framework Extension

components and possible additional dedicated physical components called *communication controllers* which are e.g. needed when implementing HW/SW connection. Note, that communication controllers are an extension to be incorporated into the generic architecture framework. An instantiation may look as illustrated in figure 10.

Given this extension one can now map the channel units functionality into different implementations either using dedicated communication controllers or distributing control and buffers among the design units only. Figure 11 illustrates that fact. It also indicates the access protocols flow of control (indicated by the dashed arrows) and data (indicated by the boldface arrows).

As examples for the alternative illustrated using the controller, one may e.g. have an implementation of an asynchronous protocol using dual-ported RAM within the controller unit. The second alternative illustrated may e.g. be a simple synchronous handshake protocol upon the bus.

2.7 Result

The output from executing the steps from above is a system design complete enough to be directly transformed into either hardware description languages like VHDL

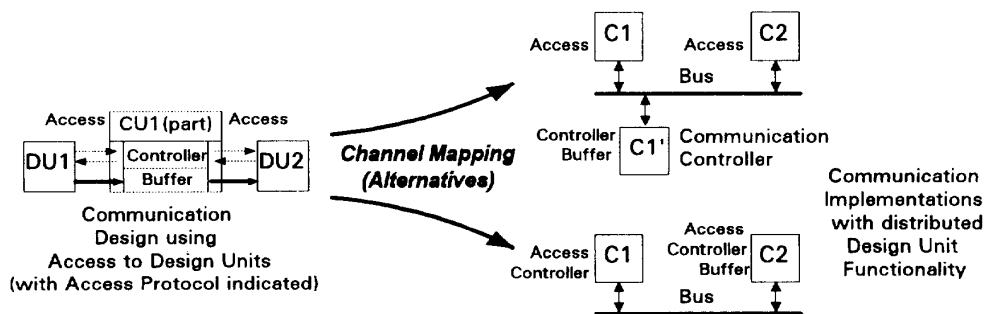


Figure 11: Channel Mapping

2.8 Recursive Process and Abstraction

It is possible however to systematically proceed to a higher level abstraction by letting design units and channel units correspond to more complex *subsystems* (we already indicated that by assigning hierarchy to design units), which can *internally* be developed according to the same process sketched above. The process of actual system design is of recursive nature in terms of the level of abstraction. The classification of whether these more complex units are regarded hardware or software within the higher level abstraction only depends on the *classification of their access*.

3. Design Process Model

relate them to one another in terms of dependencies both functional and temporal. This results in a draft version of a design process model being a framework for concrete mixed hardware/software systems engineering process standard definition.

The activities mentioned (both operational and managerial) are in the next step to be supported by tools.

4. An Example for Tool Support

As an example, the presence of the general process is briefly investigated for the hardware/software codesign environment COSMOS. COSMOS starts from the system-level specification language SDL, and produces a



heterogeneous architecture including hardware descriptions in VHDL and software descriptions in C.

Although the design starts from an SDL description, all the steps make use of SOLAR [3], a design representation language for system-level concepts. SOLAR can model system-level constructs in a synthesis-oriented manner. The basic concept of SOLAR is an EFSM that allows the representation of hierarchy and parallelism.

In SOLAR, a system is structured in terms of communicating design units. A design unit can either contain a set of other design units and communication operators known as channel units, or a set of transition tables modeled by the state table (ST) operator. This operator is used to model process-level hierarchy. State tables can be executed in parallel or serially. They can contain other state tables, simple leaf states, state transitions and global actions (exceptions). An automatic translation tool of SDL descriptions into SOLAR has been implemented. Each SDL process is translated into a design unit composed of an EFSM and a channel unit. This translation assumes that the transmission do not introduce delays.

Communication between design units is performed using channel unit. It is possible to model most system-level communication properties such as message passing, shared resources and other more complex and layered protocols.

Starting from a SOLAR representation, a system is partitioned by a system-level partitioning tool box called PARTIF [7]. This tool box allows an interactive partitioning by means of four system-level transformation primitives. The two first primitives *MOVE* and *MERGE* allow the reordering of processes hierarchy and merging processes together to form a single process. The two second primitives *SPLIT* and *CUT* allow splitting up one design unit to form inter-dependent design units for distribution purposes.

The channel binding step selects a component from a library of channel units [8]. At this level, a channel unit is described using SOLAR. The channel unit is selected in order to provide the services required by the design units. The channel mapping step selects a component from a library of channel implementations. Each channel implementation is described in C code or in VHDL. A channel implementation is selected with regard to data transfer rates, memory buffering capacity, and number of control and data lines.

This activity results in a heterogeneous description composed of a set of communication components and interconnected processes represented by VHDL for the

hardware components and C for the software modules. At this level, an implementation of channels is needed. This implementation may be the result of an early synthesis step using COSMOS or another design method.

5. Conclusion

Within this paper we outlined a terminology and general process for the design of mixed hardware/software systems. It is obvious, that the 'systems approach' taken yields a conceptual basis for hardware/software codesign.

The fields of recursive process and higher-level abstraction as well as re-use through all the phases of mixed hardware/software design could not be elaborated on in detail. The introduced theory however is a sound backbone for future work towards progress concerning these issues.

References

- [1] Voss, M., Schweizer, G.: A Development Methodology for Systems Engineering of Computer-Based Systems and its Environmental Support. Presented at: Computer Aided Systems Technology (CAST) '94 Workshop, Ottawa, Canada, 1994
- [2] Schweizer, G., Voss, M.: Managing the ECBS Process - Towards a System Theory for ECBS. Proceedings of 6th Workshop on Engineering of Computer-Based Systems, Stockholm, Sweden, Computer Society Press, 1994
- [3] Jerraya, A.A., O'Brien, K.: SOLAR: An Intermediate Format for System-Level Modeling and Synthesis. In: Computer Aided Software/Hardware Engineering, Rozenblit, J., Buchenrieder, K. (Eds.), IEEE Press, 1994
- [4] Hatley, D.J., Pirbhai, I.A.: Strategies for Real-Time System Specification. Dorset House, 1988
- [5] Voss, M.: Service-Oriented Design and its Support by a Service Access System. Internal Paper of the IMA, University of Karlsruhe, 1993
- [6] BenIsmaïl, T., et al: An Approach for Hardware-Software Codesign. To appear in: Rapid System Prototyping (RSP)'94 Workshop, Grenoble, France, June 1994
- [7] BenIsmaïl, T., O'Brien, K., Jerraya, A.A.: Interactive System-Level Partitioning with PARTIF. Proc. EDAC'94, Paris, France, Feb. 1994
- [8] O'Brien, K., BenIsmaïl, T., Jerraya, A.A.: A Flexible Communication Modelling Paradigm For System-Level Synthesis. Handouts of Int'l Wshp on Hardware-Software Co-Design, Cambridge, Massachusetts, Oct. 1993