A System Design Methodology for Telecommunication Network Applications *

Julio Leao da Silva Jr.[1,†] Chantal Ykman-Couvreur[1] Gjalt de Jong[2] Bill Lin[1] Hugo De Man[1]

{silva,ykman,billlin,deman}@imec.be jongg@sh.bel.alcatel.be

[1]IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

[2]Alcatel Telecom, F.Wellesplein 1, B-2018 Antwerpen, Belgium

Abstract

We describe a system design methodology, well-suited for telecom network applications. This methodology is being developed into a compiler called Matisse. *The entry point for this methodology is a system specification model that is first compiled into an abstract machine. The abstract machine is implementation-independent, which permits the exploration of different embedded hardware/software realizations. The proposed methodology and tools bridge the gap between system specification and synthesis tools commercially available. This yields several advantages over the design methodology currently used in industry.*

1. Introduction

Modern telecom systems are rapidly increasing in design complexity. Telecom network applications include system components for broadband networks [11], wireless infrastructures, and interactive video-on-demand servers. These and other emerging applications in similar areas are among the fastest growing segments of the system industry today.

Currently the design of telecom network applications is a collaborative effort taking place at multiple sites. Hardware and software components are designed separately, which often introduces both specification and implementation mismatches that are only detected at the final design stages. Currently, the system integration and test phases can take nearly 50% of the complete system design cycle for typical telecom network applications.

Each software component is usually specified using SDL [17]. C or C++ code is then automatically generated and compiled to machine instructions for the target processor. Run-time support is added for managing concurrency and interprocess communication [21]. Each hardware component is specified at the register transfer level, using VHDL or Verilog, and a detailed implementation is produced using logic synthesis. This level of specification is often hard to read, modify, and reuse. The code is already refined with detailed clock cycles, and specific architectural decisions are already fixed. Small changes at the system level often require very substantial changes in the specification of the hardware or software components.

*This research is sponsored in part by IWT under project "HASTEC".

†This author is supported by a Brazilian Governement Fellowship - CAPES.

Most system-level research and CAD innovations today are focussed on Digital Signal Processing (DSP) applications (e.g. [3, 7, 12]), which are different in nature from telecom network applications. Telecom network applications require complex manipulation of complex data structures that are often dynamically created and destroyed at run time, as opposed to the (multi-dimensional) array signal streams present in DSP applications that can be largely analyzed at compile time. As compile-time analysis is not possible for telecom network applications, the memory-oriented synthesis techniques proposed for DSP applications may not be used. Due to many differences in nature between these application domains, system model and design tools should be domain-specific.

DSP models tend to be data-flow oriented, which are not well-suited for control-dominated data processing behaviors found in telecom network applications that heavily rely on tight interactions between control-flow algorithms and stored data structures. The hierarchical FSM model, as exemplified in commercial systems like Statemate from iLogix, is a powerful formalism for reactive control behaviors, but it does not support well programming constructs like abstract data structures and object-oriented features. Heterogeneous design environments, like Ptolemy [3] and CoWare [16], aim to provide an open environment to smoothly integrate different models of computation together with simulation and design tools. Distributed programming languages have been proposed for programming general-purpose multiprocessor systems or distributed networks of workstations [1, 2, 4, 5, 14, 18]. While their underlying models are strongly related to our Matisse model, their implementation targets are different: they rely on elaborate run-time environments and are intended for pure software implementations. In contrast, our implementation target is intended for optimized embedded single-chip hardware/software realizations.

In this paper, we first present a system specification model, well-suited for telecom network applications, which is independent from hardware/software realizations. We then present a system design flow that permits extensive design exploration. The paper is organized as follows. In Section 2, we present the requirements to be supported by our Matisse model and by our system design flow. Then we describe the main concepts of the Matisse model in Section 3. In Section 4, after having given an overview of the system design flow, we focus on two of its major steps: the underly-

Figure 1. Requirements Example

ing abstract machine and the system architecture generation. Finally in Section 5, conclusions are presented.

2. Requirements

The requirements that are supported by our system model and design flow were obtained from actual designs [19] in the telecom network domain.

System specification requirements

Telecom network applications are conceptually seen as sets of concurrent tasks for accessing data. Therefore data have to be considered as stored objects from the beginning. Concurrency tends to be at the task level and is usually coarse to medium scale.

Although the target implementation of a telecom network application is often a mixture of software and hardware processors, telecom network applications are best-suited to be conceived at the top level from a software perspective, by modeling the system as a set of concurrent processes, possibly accessing shared data structures. Control constructs, such as *if-then-else* and *for* and *while loops*, are essential for capturing algorithmic behavior of each process.

The example illustrated in Figure 1 abstracts the requirements encountered in specifying telecom network applications and fulfilled by the proposed Matisse model. It consists of a shared linked list of records, a producer that sends data to be stored in the shared list, and a consumer that reads data from the shared list, one at a time. Once a record is read by the consumer, it is destroyed from the shared list.

While an object-oriented model is not necessary, it has been proven successful in the software design community, and it also plays a central role in large scale hardware/software system design. Object-oriented languages concentrate on the real-world entities identified in the application, which are tasks for accessing data, in telecom network applications. Object-oriented languages support data abstraction, encapsulation, polymorphism, function overloading, and inheritance, which are invaluable features in any large scale development. With these abstraction facilities, implementation decisions and low-level specification details can be hidden. This also allows easy and fast design exploration. Although the object-oriented paradigm may incur design overhead, by restricting the specification language and by automating the system design flow, these inefficiencies are minimized.

Concurrent object-oriented models are well-suited for telecom network applications, since they are intended for concurrent computations to be executed on more than one processor. Objects can encapsulate processes as well as (shared) data structures. Remote procedure calls can encapsulate interprocess communications.

Consequently, a system specification language that supports a concurrent object-oriented model is well-suited for specifying telecom network applications. Such a system specification language should also have the following characteristics: reflect the conceptual partitioning of the system, seen as a set of concurrent tasks for accessing data, be independent from the final implementation, be manipulatable to permit efficient design exploration and easily retargetable to different embedded hardware/software realizations. This is in contrast to current system specifications, which are using VHDL for specifying the hardware processors, and C/C++ for specifying the software processors.

System design flow requirements

Starting from a system specification language conforming the requirements mentioned above, the system design flow needs to be automated in order to avoid specification and implementation mismatches, to accommodate changes in the system specification, and to allow design exploration. The functional specification should be validated already at the system level, without executing both hardware and software low-level specifications, that are too time-consuming.

Also the system design flow should be optimized for telecom network applications and provide support for: refinement of abstract data types into complex data structures, such as heaps, hash tables, tress and linked lists, refinement of the virtual memory management, memory access optimization and memory synthesis. HW/SW interprocessor communication synthesis should also be provided. Such a system design flow bridges the gap between system specification and commercial tools.

3. System Model

The Matisse model, supporting the requirements presented in Section 2, is a simplified model of the one underlying Compositional C++ (CC++) [5]. CC++ is a concurrent object-oriented language extended from C++ that is intended for specifying concurrent programs to run on a network of workstations over an elaborated Operating System (OS). However, Matisse is intended for specifying concurrent processes to run on a mixture of embedded software and hardware processors over an ultra-light OS.

In the Matisse model, a system is considered as a set of concurrent tasks for accessing data. Tasks are created at the initialization of the Matisse program only, and execute concurrently. A *task encapsulates both a sequential program and local data*, on which the sequential program operates. Tasks communicate with each other *without* explicit specification of communication channels. Tasks can be mapped on physical (hardware or software) processors in various ways: in particular, multiple tasks can be mapped on a single physical processor. However, because tasks communicate with each other using the same mechanism regardless of task mapping, a Matisse program does not depend on where tasks are executed. Hence algorithms can be designed without concern for the physical processors on which they will execute.

Specifying shared data structures is also possible. With the power of Matisse to its disposal, a library of common and explicit communication concepts, like channels and buffers, can also be developed.

3.1. Concepts

Now the different concepts that are used in the Matisse model are explained in more detail.

Passive and active classes In Matisse, two types of classes are distinguished: *active* and *passive*. A passive class is identical to a C++ class. Passive objects, which are instances of a passive class, may be created and destroyed either at compile time or at run time. An active class differs from a passive class by having *its own local virtual memory space and default thread of control*. This thread is initiated at the creation of the active object, and it is specified by the special member function *body*. Active objects *may only be created at compile time*, to avoid creation of new threads at run time that may be difficult to implement on a hardware processor.

Concurrency at the task level In a typical Matisse program, the number of active objects is small, compared to the number of passive objects. The passive objects exist as data elements of active objects. Active objects are created in the main function, which yields the concurrent initiation of their bodies. Hence the main function provides the task-level concurrent structure of the Matisse program.

Communication Accessing data elements within an active object is done by using C++ pointers. It is regarded as local and hence as cheap. Active objects can be accessed by each other using *global pointers*. Intuitively a global pointer indicates both the address of the active object and the address of some data element in its local virtual memory space. Except for their potentially higher cost of use, global pointers are used just like C++ pointers.

Inside a thread, computation can be executed in another active object via a Remote Procedure Call (RPC) and data can be communicated between both active objects, by using global pointers.

Synchronization Due to concurrent computations, several accesses to data elements or member functions in an active object can occur simultaneously. Matisse provides one method for controlling the order in which things happen, by using atomic functions. Whenever several threads are calling an atomic function, this atomic function is executed the required number of times in a sequential order. Also the execution of an atomic function never interleaves with the execution of another atomic function of the same active object. This concept of atomic function is based on the monitor concept introduced by Hoare in [10]. Since there are no condition variables, some rules for defining atomic functions must be followed in order to avoid deadlocks.

Shared Data structures Each active object represents one local virtual memory space, with a default thread executing in it. Passive objects are never shared between active objects. If it appears that a passive object needs to be shared by several active objects, the designer has several options. He can, for example, specify typical active objects with an empty *body* function and whose data elements are those passive objects to be shared. The protection of the passive objects, using atomic functions, is the responsibility of the active object itself [22].

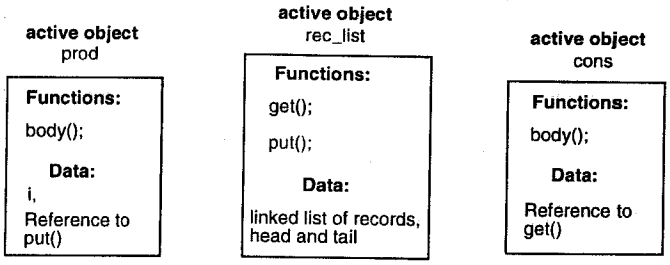

Figure 2. Example in Matisse

3.2. Example

The simple example, introduced in Figure 1, can be specified, using Matisse, as follows.

```
class record {
public:
  int     item;
  record* next;
  void*   operator new(size_t);
  void    operator delete(void*, size_t);
};

active class consumer {
public:
    body (shared_list* global gp) {
      int i;
      i = gp->get();
      cout << i << endl;
    }
};

active class producer {
    int i;
public:
    producer () { i = 0; }
    body (shared_list* global gp) {
      int i;
      gp->put(i);
      i++;
    }
};

active class shared_list {
   record *head, *tail;
public:
   shared_list () {
      head = 0;
      tail = 0;
   }
   atomic int get () {
      // return first record of the list and then delete it
   }
   atomic void put (int i) {
      // store i at the end of the list
   }
};

int main () {
  shared_list* global rec_list;
  producer* global prod;
  consumer* global cons;

  rec_list = new shared_list();
  prod = new producer(rec_list);
  cons = new consumer(rec_list);
}
```

This is illustrated in Figure 2. The shared linked list of records is specified by the object `rec_list` of the active class `shared_list`, the producer is specified by the object `prod` of the active class `producer`, and the consumer is specified by the object `cons` of the active class `consumer`.

Through the definition of the `new` and `delete` operators, the memory management can be defined precisely, for instance to limit the number of records and the maximum size of the shared list.

The tasks executed by producer and consumer are specified by their `body` member functions. Insertion and deletion of records in the linked list are specified by the member functions `get()` and `put()` of `shared_list`, which are called as remote procedures by the producer and consumer objects. In this way, neither the producer nor the consumer have to worry about the management and the access control of the shared memory.

4. System Design Flow

A systematic approach to design a system consisting of concurrent processes accessing shared data must provide mechanisms to evaluate alternatives and to reduce the cost of backtracking from bad choices. Implementation-independent issues must be considered early and implementation-dependent aspects, such as system and memory architecture, must be delayed until late in the system design flow. Before being specified using Matisse, the system to be designed must be first analyzed, then shared data must be identified, in order to partition the system into a set of concurrent processes for accessing these data.

We propose the system design flow depicted in Figure 3, starting from an initial concurrent object-oriented specification within the Matisse model, and targeting a heterogeneous implementation of software and hardware processors. This refines the general design methodology for telecom network applications introduced in [13].

The *Matisse program*, using abstract data types, as sets and association tables, specifies the system to be described. The example, shown in Section 3.2, is such a Matisse program. The *abstract machine* consists of an internal representation, which characterizes the Matisse program as a network of communicating virtual processing objects.

System architecture generation consists in allocating a number of hardware and software physical processors and mapping the abstract machine to this target architecture of physical processors.

Software processor synthesis consists in generating the complete specification of each software processor, so that synthesis is made possible by using traditional software design tools for code generation. *Hardware processor synthesis* consists of memory synthesis and VHDL code generation, so that synthesis is made possible by using traditional hardware/behavioral synthesis tools. Interprocess communication is also refined in each HW/SW processor synthesis, into intraprocessor communication and interprocessor communication, which then can be synthesized, using the system integration tool-box *CoWare* [16].

In this paper we focus only on the abstract machine and system architecture generation. Both steps are described in the following subsections.

4.1. Abstract machine generation

The goal of this step is to generate an abstract machine, from the initial Matisse program, to be used as internal representation through the whole system design flow. This abstract machine allows efficient system design exploration interactively with the designer and it is still independent from

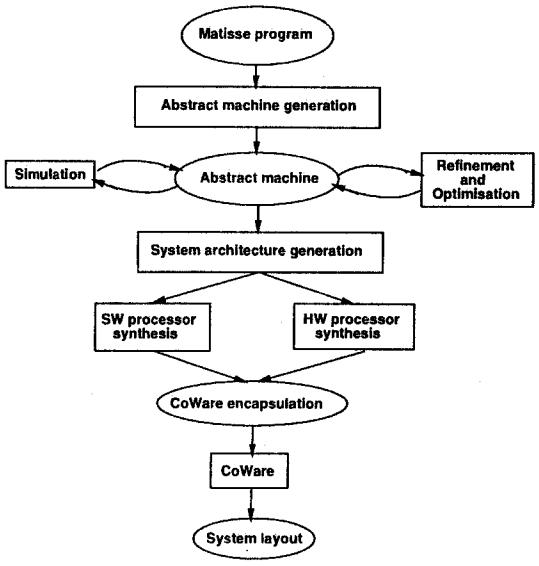

Figure 3. Matisse **Design Flow**

the final HW/SW realization. For telecom network applications, system design exploration is needed for the following purposes: implementation of abstract data types into efficient complex data structures [24], memory management of these complex data structures, which are dynamically allocated and deallocated by concurrent processes [6], memory access optimization yielding ordering of concurrent threads [23], and exploration of different embedded HW/SW realizations, based on interprocess communication costs.

In the Matisse compiler, the abstract machine is characterized as a network of communicating Virtual Processing Objects (VPOs), managed by an ultra-light OS. Each VPO is in a one-to-one correspondence with an active object of the Matisse program. This is illustrated in Figure 4. A VPO consists of threads mapped on it and of a Virtual Local Memory (VLM) to store the data elements and member functions mapped on it. The threads mapped on a VPO are: (1) the default thread, specified by the *body* function of its associ-

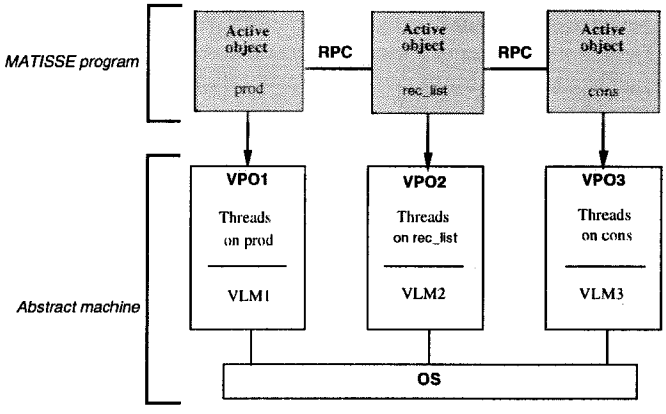

Figure 4. Abstract machine of the Matisse **program example**

ated active object, and (2) other threads, initiated by *body* functions of other active objects.

For our previous example, the abstract machine consists of three VPOs. The VPO associated with `rec_list` executes threads specified by `i = gp->get()` and `gp->put(i)`, whereas the VPOs associated with `prod` and `cons` execute threads specified by the corresponding *body* functions.

This abstract machine is simulatable, so as to help the designer in efficiently driving the system design exploration, and to functionally validate any refinement in the Matisse specification.

With the abstract machine, a control dominated flow graph (extended from the model defined in [9]) is also associated in view of memory access optimization.

Abstract machine specification The abstract machine is specified as one C++ program with calls to basic functions defined in the available multi-thread library [20]. Such a C++ program is derived from the Matisse specification, as follows: consider each specified `active class` as a derived class of one base class `active`; consider all `global pointers` as classical `C++ pointers`; define a semaphore in each active class containing atomic functions; expand all atomic functions, to lock and unlock the corresponding semaphore; dynamically schedule all concurrent threads on the queue of the OS; expand the Matisse main function, to concurrently initiate the `body` function of each created active object.

This C++ program can then be compiled using any C++ compiler and the resulting executable permits to simulate the abstract machine on any available computer.

Refinement As previously explained, refinements of the initial specification are needed and related to abstract data types, bit true behavior and virtual memory management. Abstract data types are refined by implementing derived classes, based on cost functions for area and power, as functions of the size and access number. Virtual memory management is refined by implementing optimized virtual memory managers and operators `new` and `delete` as follows. All data structures to be mapped on a VLM, are assigned to Virtual Memory Segments (VMS). A VMS is a collection of frames, each frame being a chunk of storage large enough to store one instance of a specific data structure. A virtual memory manager is associated to each VMS, to create or delete data structures at run time and in a concurrent way, and to keep track of the used and freed frames of the VMS.

Optimization Run-time interprocessor communication conflicts may lead to significant decrease in real-time performance. Therefore it is necessary to impose minimal ordering constraints between the behaviors associated to each physical processor so that the real-time requirements of the overall system are met.

Whenever possible, threads running on a single processor should be statically scheduled and interleaved to minimize the run-time overhead and to guarantee the required performance, mainly in terms of available memory bandwidth. Communication overhead can be enormous but can be reduced to its absolute minimum by scheduling the different threads in a global way. Thread interleaving happens during data flow analysis, on the control dominated flow graph, derived in the abstract machine.

The default threads, corresponding to bodies of active objects, mapped on hardware physical processors must be deterministically ordered. This is necessary to ensure correct dependence order between threads. Due to the resulting sequential behavior of the hardware physical processors, all conflicting memory accesses can also be predicted at compile time, and a synthesis methodology can be developed to properly order them within a given cycle budget, and to trade-off the memory bandwidth against minimal area and power [24], yielding an optimized distributed memory. This is done in the physical memory management, in the hardware processor synthesis.

4.2. System architecture generation

The goal of this step is to generate the system architecture, by allocating a number of hardware and software physical processors and by assigning the VPOs of the abstract machine to the target physical processors.

System architecture skeleton The skeleton of the system architecture is defined as a network of a number of hardware and software physical processors and possibly of shared memories. This skeleton is defined interactively by the designer. In many telecom network applications, many concurrent tasks have to operate on the same data and to communicate with each other. In order to yield a feasible solution, a possibly distributed shared memory architecture is required. In our target architecture model shared memories are assigned to a physical processor which mainly consists of a local memory, and almost no behavior.

Virtual processing object assignment The VPOs from the abstract machine are currently interactively assigned to the target hardware/software physical processors and shared memories by the designer, for the following reasons. First, the number of VPOs is usually rather limited. Also, research on automatic partitioning is still under investigation (see [8, 15]) and it is not yet known whether it will offer optimal solutions. Finally, VPO assignment can have a large impact in the overall communication cost and the system performance due to run-time interprocessor communication conflicts.

Physical processor skeleton Each physical processor consists of a *core* that executes all concurrent threads from the VPOs assigned to the physical processor; a *memory* that stores data structures and implementations of all needed member functions and an *I/O unit* that is responsible for interprocessor communication, based on sending and receiving data via an abstract communication channel. This architecture is illustrated in Figure 5.

The virtual memory of a physical processor is characterized in terms of VMSs, resulting from the virtual local memories of the VPOs, assigned to the physical processor. The implementation of the different parts of the physical processor depends on whether it is software, hardware or shared memory. This is dealt within the software and hardware processor synthesis steps of the design flow of Figure 3.

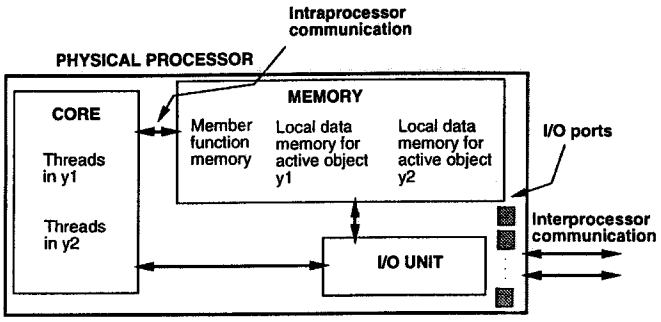

Figure 5. General architecture of a physical processor

Processor communication refinement After the assignment of the VPOs, communications between VPOs assigned to the same physical processors are refined into intraprocessor communications. This means for example that for VPOs mapped on the same software physical processor, accesses by global pointers behave as simple C++ pointers. Communications between VPOs assigned to different physical processors are refined into interprocessor communications. Three types of interprocessor communications are possible: read remote data, write data into a remote processor and ask a remote processor to execute a piece of computation, where data can be either a simple data type or a complex data structure. Interprocessor communication is specified, using CoWare communication operations and the channel layout derived from the initial Matisse program.

After abstract machine and system architecture generation, the synthesis of hardware and software processors can proceed, as depicted in Figure 3.

5. Conclusions

First we presented the requirements that must be supported by a system compiler to design telecom network applications. Those requirements were obtained from actual designs [19]. Then, we introduced the main features in the Matisse model that satisfy the requirements for capturing telecom network applications. Finally, we proposed a system design flow, suitable for this kind of applications. We focussed on two major steps, the abstract machine generation, implementation-independent, and the system architecture generation, implementation-dependent. The advantages of Matisse over the design methodology currently used in industry and the missing aspects in commercially available tools were also presented.

In the future, we want to validate our design flow on other different examples and possibly in different fields of application. We also intend to investigate on how to include timing constraints in the system specification.

Acknowledgments This work is part of a joint collaboration between IMEC and Alcatel-Bell. We would like to thank Alcatel for their contributions and support in this project. We also would like to thank K. Croes, M. Miranda, E. Umans, S. Vercauteren and D. Verkest. for numerous insightful discussions.

References

[1] H. Bal, M. Kaashock, and A. Tanenbaum. Towards a Method of Object-Oriented Concurrent Programming. *IEEE Trans. on Softw. Eng.*, 18(3), 1992.

[2] A. Black, et al. Object Structure in the Emerald System. In *ACM OOPSLA Conf.*, pp. 78–86, 1986.

[3] J. Buck, et al. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. Technical Report, University of California, Berkeley, August 1992.

[4] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Comm. of the ACM*, Sep 1993.

[5] K. M. Chandy and C. Kesselman. *CC++: A Declarative Concurrent Object-Oriented Programming Notation.* MIT Press, 1993.

[6] G. de Jong, et al. Background Memory Management for Dynamic Data Structure Intensive Processing Systems. In *Proc. ICCAD*, 1995.

[7] H. De Man et al. Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms. In *Proceedings of the IEEE*, vol. 72, no. 2, pp. 319–335, February 1990.

[8] R. Ernst, J. Henkel, and T. Benner. Hardware-Software Cosynthesis for Microcontrollers. *IEEE Design and Test of Computers*, pp. 64–75, Dec 1993.

[9] M. Eyckmans, E. de Greef, and P. Petroni. Information Model for the BB/OAT Flow Graph Environment. Technical Report, IMEC, June 1994.

[10] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Comm. of the ACM*, 17(10):549–557, 1974.

[11] B. Kumar and J. Ranade. *Broadband Communications, A Professional's Guide to ATM, Frame Relay, SMDS, SONET and B-ISDN.* McGraw-Hill Series on Computer Communications, 1994.

[12] R. Lauwereins, et al. Grape-II: A System-Level Prototyping Environment for DSP Applications. *IEEE Computer*, pp. 35–43, Feb 1995.

[13] B. Lin. A System Design Methodology for Software/Hardware Co-Development of Telecommunication Network Applications. In *Proc. DAC*, 1996.

[14] B. Meyer. Systematic Concurrent Object-Oriented Programming. *Comm. of the ACM*, Sep 1993.

[15] R. Niemann and P. Marwedel. Hardware/Software Partitioning Using Integer Programming. In *Proc. ED&TC*, pp. 473–478, 1996.

[16] I. Bolsens, et al. Hardware-Software Codesign of Digital Telecommunication Systems. In *Proceedings of the IEEE*, April 1997,

[17] R. Saracco and P. Tilanus. CCITT SDL: An Overview of the Language and Its Applications. *Computer Networks and ISDN Systems, Special Issue on CCITT SDL*, 13(2):65–74, 1987.

[18] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling.* Wiley, 1994.

[19] Y. Therasse, G. Petit, and M. Delvaux. VLSI Architecture of a SDMS/ATM router. *Annales des Telecommunications*, 48, 1993.

[20] *USL C++ Language System Release 3.0 Library Manual.* UNIX System Laboratories, Inc., 1992.

[21] E. Verhulst. Virtuoso: Providing Submicrosecond Context Switching on DSPs with a Dedicated Nano Kernel. In *Proc ICSPAT, Santa Clara*, Sep 1993.

[22] P. Wegner. *Design Issues for Object-Based Concurrency.* Springer-Verlag, 1992.

[23] S. Wuytack, et al. Flow Graph Balancing for Minimizing the Required Memory Bandwidth. In *Proc. of the Int. Symp. on System Synthesis*, pp. 127–132, 1996.

[24] S. Wuytack, F. Catthoor, and H. De Man. Transforming Set Data Types to Power Optimal Data Structures. *IEEE Trans. on CAD*, 15(6):619–628, June 1996.