Designing Digital Video Systems: Modeling and Scheduling

H.J.H.N. Kenter[†], C. Passerone[††], W.J.M. Smits[†], Y. Watanabe[‡] and A.L. Sangiovanni-Vincentelli[‡]

[†] Philips Research Labs, Eindhoven, Netherlands, [††] Politecnico di Torino, Turin, Italy
[‡] Cadence European Labs, Rome, Italy

Abstract

An advanced Digital Video Broadcasting (DVB) system is used as a design driver for an IP-based real-time design methodology explored in the ESPRIT/OMI COSY project [3]. The design methodology is supported by the Felix VCC environment [10], provided by a COSY partner Cadence, and tool-set developed for COSY. In this paper, we focus on two key aspects of the design: behavior modeling and code generation. For the behavior modeling, we present the model of computation used to represent the DVB and the technique for expressing this particular model with the more general model of computation supported by the Felix technology. In a companion paper [4], the architecture selection and communication refinement are described. Once the architecture is selected and a partitioning has been decided, the implementation phase starts. In this phase, for most system designs, a great deal of software has to be written to "customize" the programmable components of the architecture. Obtaining an optimized and correct-by-construction software implementation is fundamental in an effective design methodology. Here we focus on a software generation technique which aims to reduce run-time overhead for functions executed on a single CPU, by generating a minimal number of run-time tasks.
keywords: System design, IP integration, Software generation

1 Introduction

IP-based design methodologies have been discussed as a promising vehicle to realize cost-effective design practice for real-time embedded systems. A goal of the COSY project [3] is to finalize a general IP-based system design methodology supported by the Felix VCC environment and tool-set to a consumer electronics application characterized by a great deal of data-flow processing and control. We use a Digital Video Broadcasting (DVB) system designed by Philips Semiconductors as a design driver to assess the methodology. The objective behind this choice was to use a non-trivial design, rather than a toy example, based on commercial products, that possesses important characteristics commonly observed in state-of-the-art real-time embedded systems. Two such characteristics are (1) the design is neither control dominated nor data-flow dominated but rather a mixture of both, and (2) the design is the basis of a new generation of existing products, and as such, requires to re-use parts of the existing designs while meeting new specifications. Our system takes as input MPEG2 transport streams and decodes them, and can resize and filter the decoded objects under the control of real-time commands issued by a user.

The design methodology is based on a set of successive refinement steps starting with the specification of the behavior of the design and of the constraints it has to obey. Fundamental to this step is the choice of a model of computation. A model of computation for a given application domain is often chosen so that key properties for designs of that particular domain are ensured by construction. However, if we wish to obtain a design methodology that is general enough to support a fairly large number of application domains, the underlying model of computation should be able to express almost all models of computation. In Felix, the model of computation is based on the network of Co-design Finite State Machines [1], which is low level enough to represent other computation semantics. In the design of the DVB systems, on the other hand, the model used is at a much higher level. Hence, in order to capture these systems in Felix, it has to be described in terms of the semantics provided in Felix. This process is quite interesting in that it exposes a general problem for system design. We believe that our solution is simple and at the same time powerful enough to be captured in the environment so that all future designs in this domain can now be expressed in the higher level model without worrying about the translation into the low level model. The model of computation introduced for the DVB system is called YAPI and allows mix-and-match of several IP blocks guaranteeing correctness of the composition.

The analysis of the functional aspects of the DVB system has led to the choice of an appropriate library of IP blocks whose granularity is the result of an informal optimization step that trades-off re-usability with efficiency. We present the IP library that has been used to define the functionality of the DVB system.

The architecture is defined using the COSY architecture model as a set of interconnected components that composes the implementation of the design. The designer then explores different implementations of the behavior on the architecture. Here, trade-offs are made between cost and performance, by reducing run-time overhead to meet real-time constraints under the available resources. Three key issues concerned in this step are: partitioning between hardware and software, communication refinement (e.g. bus protocols, buffer sizes), and generation of run-time tasks for software [1]. A partitioning is obtained by defining a mapping of the behavior onto the architecture, i.e. deciding which architecture components will execute the behavior IPs. The mapping is then elaborated by communication refinement and task generation. Once the behavior and the architecture have been imported into Felix, a mapping can be defined graphically and communication is refined by specifying protocols and by setting implementation parameters.

The generation of run-time tasks is supported by a software generation technique developed in COSY. It generates a minimal set of run-time tasks for a behavior mapped to a CPU running a multi-tasking RTOS, in order to reduce run-time overhead, e.g. interrupts or synchronization, as much as possible. It schedules

[1] Scheduling of the generated tasks and implementation of each task will follow this process.

each run-time task and generates C code for it. The result of code generation are then incorporated into Felix, and its cycle-approximate simulator is applied to verify the overall performance of the design.

The paper is organized as follows: In Section 2, we present the DVB system. Section 3 is concerned with the behavior modeling. We first present YAPI, with the focus on the underlying computation semantics. We show that this API is suitable for DVB applications, in which data loss must be prevented in dataflow processing and a special attention is required for interaction between dataflow and control processing. We then describe how to import this behavior into Felix. Section 4 presents the software generation technique outlined above.

2 The Design

Digital (uncompressed) video systems are characterised by high speed and high throughput processing, in which operations are repeatedly executed. On the other hand, compression in the digital video domain (MPEG) also requires control processing, since compression/reproduction of data is controlled based on data being handled. Control processing is also needed to support real-time user interaction. Thus, both data and control flow aspects and their interaction need special attention when modeling such a system.

In general, the systems receive MPEG2 transport streams, where the user selects channel(s) to be decoded. The associated video or graphic objects are then descrambled, demultiplexed, and decoded. The user may also define post-processing operations on the decode objects, such as filtering, zooming, and composition. In the COSY project, we focus on MPEG2 video decoding with picture in picture capabilities. We have developed an IP library which enables designs of three basic applications: HDTV, QUADTV, and PIPTV, which consists of the following:

- the MPEG2 Transport Stream (ISO/IEC 13818-2) demultiplexer. This TSDEMUX function extracts from an incoming Transport Stream (TS) those Packetized Elementary Stream (PES) packets that correspond to the Packet IDentifiers (PID) selected by the user.
- the MPEG2 Packetized Elementary stream header parser. This **PESPARSER** function parses the incoming PES packets to collect Elementary Stream (ES) data per PID.
- the MPEG2 decoder. The H.262 compliant MPEG2 video bitstream decoder **MPEGDECODE** decodes all video ES streams up to main profile and high level (MP@HL).
- the Resizer. The **RESIZE** function deals with (user controlled) scaling images in the range of 0.16 to 10, both horizontally and vertically. For scaling we use horizontal and vertical sample rate conversion and implement them by 6 tab/64 polyphase filters. A simplified Producer-Filter-Consumer model reflecting this functionality is worked out in this paper and in the complementary paper [4].
- the Image Controller. **IMAGECONTROL** combines a number of arbitrary sized video images into a single new image. For all the input images position and overlay priority are controlled by the user.
- the User Controller. The **USERCONTROL** provides the user with an interface to control his application. Upon changes of the user settings, it calculates and sends control data to the several building blocks in the application.

An example of the PIPTV application is depicted in Figure 1. The application is capable of demultiplexing an input MPEG2 TS stream selecting two PIDs into a PES stream. It extracts the two MPEG ES streams from the PES stream, decoding them at the main profile and main level into two Standard Definition (SD) video streams. Further, it can resize one SD stream by a variable ratio (both horizontally and vertically), and composes the SD video stream and the resized video stream to a Picture in Picture SD image with a controllable PIP position.

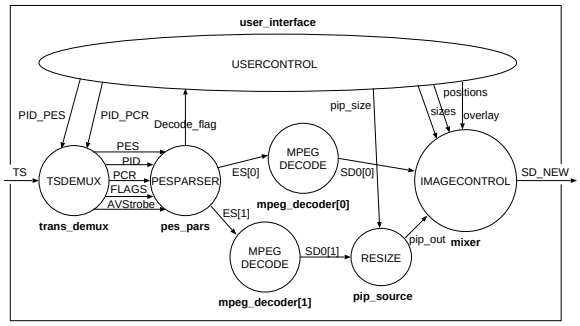

Figure 1: The PIPTV application.

3 Behavior Modeling

3.1 Application modeling: YAPI

The API has been defined with the following main goals. First, it is easy to compose IPs for DVB applications. This is done by specifying potential parallelism explicitly and by ensuring loss less communication of data: two characteristics commonly observed in this domain of applications. It also supports functions often used in interaction between control and data processing. Second, the specification retains the possibility of either hardware or software implementation, or both. Third, it is easy to deal with abstraction. This is important in IP based design, since except for the interface, the user of an IP does not need to or even cannot know all the details of the (third party) IP. The API allows one to specify a behavior of an IP, independent of how data is provided to the IP.

The model is an extension of the Kahn Process Network model [6] and we call it Y-chart [7] Application Programmers Interface (YAPI). The model consists of:

Processes A process is made of a process function and a set of input and output ports through which it communicates with other processes (or environment). The function is defined by C, together with three constructs to support operations on ports: `read`, `write`, and `select`. To avoid data loss, `read` and `write` block when data is not available or cannot be delivered, respectively. `select` also provides another blocking mechanism. It takes two input ports as input and returns a port ID. If neither input port has data available, it blocks. It identifies the port that has data available. If both ports do, `select` chooses one of them non-deterministically, because we do not want the programmer to influence the process scheduling, when specifying the applications. This non-determinism will be resolved later at the mapping level, where the designer chooses a particular deterministic implementation by taking into account various design and architecture constraints.

Directed fifos Via the input port of a fifo we can store data in the fifo and via the output port retrieve data from it.

Process network Processes are connected port-to-port by means of fifos, forming a network that is known at compile time and does not change at run time. Each output port of a process is connected to precisely one input port of a fifo and v.v.

A network of processes connected by fifos has deterministic behavior if and only if `select` is not used. If it is, behavior will be non-deterministic, until a scheduling of processes is determined. As an example we consider a Producer-Filter-Consumer configuration: a Producer sends video data to a Filter, which processes it using a set of coefficients and sends the new data to a Consumer (data path). The filter coefficients depend on a user-controlled resize factor (control path). We model the interference of the control path and data path in the filter process with a select method on both input ports. If neither input ports provide data, the process blocks. If only one of them provides data, this

is deterministically indicated by the return value of the select. If both data are present, one of the ports is non-deterministically indicated. The designer still has all the freedom to react on the return value of the select. The return value *can* be used to (non-deterministically) switch between execution of the data path and execution of the control path. On the other hand he can also model to always do the data path and if needed do the control too.

3.2 Behavior Capturing in Felix

In this section we describe how the IP library and the application defined in Section 2 are imported into Felix, thus re-creating the IPs in Felix.

3.2.1 Felix Model of Computation

Felix function diagram is a network of functional blocks connected through ports, and the model of computation (MoC) is based on Discrete Event simulation, or "fire and exit": a functional block is activated on receiving a token on a port, it can post tokens to ports during its execution, and when it is finished the token which activated the block is discarded. The blocking write, read and select semantics of Section 3.1 must be realized on top of this semantics.

The Felix tool supports several languages to express behaviour. We use ECL [8]. It is an extension of Standard C, offering additional constructs: `PAR`, `emit`, `await`, `present`, `abort`. The semantics are those of the Esterel primitives `||`, `emit`, `await`, `present`, and `abort` [2]. In short, `emit` and `await` offer non-blocking write and blocking read, `present` tests signal presence, and `abort` allows abortion of the execution of sections of code. The attractive feature of `await` is that it offers blocking semantics to the user, even though the underlying functional block in Felix exits in order to allow other blocks to run. This requires state preservation, which is looked after by ECL.

The behaviours we want to import into Felix have been defined using YAPI, which not only provides blocking `read`, as does ECL, but also blocking `write` and `select`. To enable the import of such behaviours into Felix, we first create a functional block for each process, and convert the behaviour to ECL. Since ECL extends plain C and YAPI is also based on C, this boils down to translating the blocking `read`, `write` and `select` of YAPI. As processes are connected through fifos, also in the Felix functional diagram we connect functional blocks using fifos by creating a functional block for each fifo.

3.2.2 Modeling YAPI semantics in Felix

The original behaviour uses YAPI's `write`, `read` and `select`. We present here how they can be implemented using three ECL primitives: `emit`, `await` and `present`.

Blocking write can be achieved in two ways: one is first to wait for a permission to send data and then send it once a permit is given, while the other is first to send data and then wait for an acknowledge. This implies a protocol in which also the reading process emits a token, which corresponds to either a permit or an acknowledge. When the writer and reader are connected directly, the order of events is important because of the way Felix and ECL operate: an incoming token is seen only when a block is `await`ing it; else it is lost.

To prevent such a token loss, we use a scheme that allows both the writer and reader to initiate a transaction, i.e. either of them may be the first to send a token. The adopted solution is to put a block in the middle of the writer and reader. This block is passive in the sense that it does not take the initiative on either side but responds to incoming requests. At the writer's side, the request is the data to transfer. At the reader's side, it is a request for a data item. The block always accepts incoming data. It does not acknowledge the data until room becomes available. See Figure 2.

The `select` primitive uses another feature ECL inherits from Esterel: the argument of `await` is really a signal expression. Hence, we can await the occurrence of one or both of two signals and then

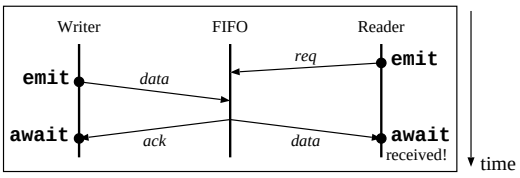

Figure 2: Time lines for req/ack protocol

test which one is active. There is a subtle difference in programming models: in YAPI we express that we want to wait for data on one of two channels and then read it; in ECL the `await` statement already reads the values of the active signals and we test afterwards with the `present` construct which ones we did read. Figure 3 shows how we can translate a YAPI fragment with `select` to ECL, disregarding the req/ack protocol.

```
n = select (in1, in2);
if (n == 0) {
  read (in1, x);
  f1 (x);
} else if (n == 1) {
  read (in2, y);
  f2 (y);
}
```

a.

```
await (in1 | in2);
present (in1) {
  x = in1;
  f1 (x);
} else present (in2) {
  y = in2;
  f2 (y);
}
```

b.

Figure 3: a. YAPI fragment, b. ECL counterpart

The code fragment to the left awaits data on either of two input signals, blocking if none is present, using a `select` statement. It uses the return value to determine which input has data available, which can then be read. The ECL code to the right reflects the blocking on two input signals, but when it terminates all data present on the inputs is consumed and available in `in1` and/or `in2`. If both inputs have data, we use only `in1`. Thus, the non-determinism of Section 3.1 turns into priority for the first input.

3.2.3 FIFOs

In Section 3.2.2 we argued the need for extra blocks in the Felix functional diagram to model blocking semantics, by means of a request/acknowledge protocol. In fact, these blocks model in an explicit way the channels of the original functional behaviour, and we bind the FIFO behaviour that we associate with channels to these blocks. Thus, FIFOs become explicit functional blocks in Felix, connected to their writers and readers by four channels. Figure 4 shows this for a Producer-Filter-Consumer example.

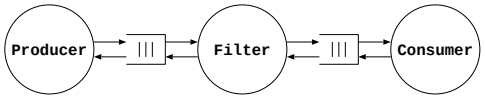

Figure 4: Producer-Filter-Consumer with FIFOs

4 Software Generation

After the behavior of the design has been captured and simulated to check its correctness, it is mapped onto an architectural diagram. An architectural diagram is a collection of entities (microcontrollers, DSPs, ASICs, memories and so on), which communicate through dedicated or shared buses. Each block in the functional diagram should be mapped to an architectural entity, and communication links between blocks assigned to different entities should be mapped to buses. More details about architecture capturing are described in the companion paper [4].

This mapping process is not automated in our design methodology, but is done by the designer. The mapping is not one-to-one in general, and may assign more than one process to the same architecture component like a CPU or a DSP. Therefore, a proper scheduling is required for those processes mapped to the same processor. This section describes code generation for the software partition of a design, or in case of multiprocessor systems, the software partition per processor. Since in these cases the computational resource is shared by the processes, all parallel operations should be serialized[2], either at compile time or at run time. This decision has a big impact on the performance of the final implementation, and thus it is an important design parameter. In the COSY project, we provide a procedure aiming to reduce run-time overhead as much as possible. Specifically, the procedure takes a set of processes mapped to a single processor, and then

1. generates a minimal number of run-time tasks,
2. schedules each task at compile time whenever possible,
3. generates an efficient C code for the schedule of each task.

Extensive research has been made on scheduling algorithms, of which the two most popular classes are static scheduling, e.g. [9], and real-time scheduling, e.g. [5]. The former makes all scheduling decisions at compile time, thus reducing run-time overhead completely. However, not all applications can be modeled in this way, since information needed for scheduling is often only available at run time. A data-dependent choice is an example often encountered in control processing. Real-time scheduling can handle data dependencies at run time, but it seldom considers communication patterns. Especially in data processing, data samples are often generated with known patterns on rates and latencies, and this information can be used to reduce run-time overhead.

The single-processor software scheduling technique that we are developing combines the best aspects of these approaches. It schedules basic blocks at compile-time, i.e. statically, serializing concurrency and resolving multi-rate dependencies. Further, sets of basic blocks that are connected with data-dependent choices or related by synchronization are grouped into a run-time task. C code is generated for each such task, which implements the schedules of the basic blocks and uses `if-then-else` constructs for data-dependent choices. In this way, data-dependent operations can be included in compile-time schedules, leaving the actual resolution of dependencies to run time. Synchronization of basic blocks within a single task is resolved in the code, saving precious RTOS synchronization resources and memory. The resulting tasks are scheduled at run time using RTOS-supplied synchronization and context switching primitives. We call this technique *Quasi-static scheduling* (QSS), since it tries to schedule processes as much as possible at compile time.

This technique is particularly interesting in DVB applications, where high speed data flow coexists with control decision, both synchronous to the bitstream, as compression/decompression, and asynchronous, like user interaction with a set-top-box. The Producer-Filter-Consumer system, described in Section 3.1, is a typical example of such an application: a completely static schedule would not be able to handle control, while current real-time scheduling techniques would impose an excessive overhead in the data flow part. Our approach is different because we can model reactions on "soft real-time" control, having a minimal cost penalty for the "hard real-time" data path, thus yielding an efficient implementation.

To realize this, we need a mathematical model with which the system function can be modeled, as well as scheduling algorithms that work on it. We also need a mechanism that translates a language specification of the system function into the model.

For the model, we use a class of Petri Nets (PNs) called Unique Choice Petri Nets (UCPNs). PNs are bipartite graphs with two types of nodes called *places* and *transitions* [11]. An advantage of PNs over other models of computation is that it naturally models both control and data operations: the former mainly consists of data dependent choices, the latter are series of mixed rate operations with intensive concurrency. Both choices and concurrency can be conveniently modeled using the two types of nodes. Another advantage of PNs, unlike general Data Flow models, is that the question on schedulability is decidable, and therefore it is possible to algorithmically find a schedule, if one exists.

A PN is said to be *unique choice* if and only if at any time a place with multiple successor transitions is marked, either only one successor transition is enabled, or all its successor transitions are enabled. Thus, a choice is either not a choice or a free choice. A free choice represents a data dependency that must be resolved at run-time. UCPNs are deterministic by definition, i.e. a transition to fire is uniquely determined once information is provided at run-time to resolve the data dependencies. This property makes it easier to develop scheduling algorithms, with the only restriction that the system function represented by the model must be deterministic under given data values, which is the case for embedded systems specified in deterministic languages like C.

In our model, it is relatively straightforward to translate a large portion of a C-based language to it. This allows many parts of legacy designs, either for creating design libraries or for other design projects, to be imported into our design flow with minor changes. Examples are the building blocks of Section 2. This is a key advantage, since reuse of previously developed IPs is one of the main factors to increase designers' productivity, and hence reduce design time.

As a specification language, from which a UCPN is derived, we use YAPI with a restriction; we exclude the use of recursion. In the translation from YAPI to a UCPN, the control flow and ports are modeled by places, while the C statements are associated with transitions. Successive transitions can be merged to reduce the net, if they are not input or output of a place representing a port. The same applies to the *then* and *else* branches of a data dependent choice, so that the choice disappears from the PN, but lives as a piece of C code associated with the resulting transition.

The translation algorithm guarantees that the resulting PN is Unique Choice, except for the `select` construct. As defined in Section 3.1, the `select` construct non-deterministically *selects* one of the two ports to which this construct is applied, if both ports have data available. Therefore, non-unique choiceness arises in the resulting PN. However, this PN can be made Unique Choice if a priority is defined over the ports so that in case both ports have data, one with the higer priority is selected. Such a priority is defined by the designer as a design parameter at the mapping step, and since the proposed technique is applied after the mapping has been made, one can incorporate these priorities into the model.

This translation algorithm is used to generate a UCPN for each process, and the UCPNs for two processes communicating through a pair of ports are connected by merging the places for those ports.

An example of a fragment of UCPN representing the Producer-Filter-Consumer system is shown in Figure 5, where the higher priority is given to the port for coefficients. To make the net Unique Choice, we need a pair of complementary places for the coefficient port to distinguish the presence and absence of a token at the port.

Once a UCPN is obtained, it is analyzed to find a schedule. This operation involves three steps:

1. determine if the UCPN is schedulable in finite memory,
2. if so, find a schedule that minimizes the number of tasks,
3. generate C code for the tasks based on the schedule.

For the first two steps, we extend scheduling techniques developed for a sub-class of UCPNs [12]. The result is a set of sequences of transitions for each task, with a guarantee that no matter how data dependent choices are resolved at run-time, there exists a sequence in the set that can be executed with finite memory so that the system state returns to the initial state after the execution.

The last step produces the final C code for each task. This is done by stitching the statements of the original C code associated with transitions based on the sequences computed for the task.

[2] Our current approach considers a CPU as a single processor which executes operations sequentially, even though many of the modern microprocessors support out-of-order or VLIW execution.

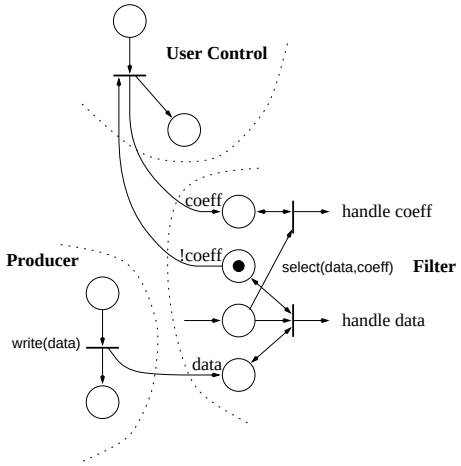

Figure 5: UCPN for the Producer-Filter-Consumer

The synthesized code can then be read back in Felix, where the processes originally mapped to the processor are replaced by the run-time tasks generated from the UCPN, and simulated along with all the blocks mapped to other processors or hardware. The code is of final implementation quality, since the scheduling has been optimized and functions are written by designers.

5 Summary and Acknowledgments

We presented the application of a general system design methodology to the design of a DVB system. A model of computation has been presented that is optimized for the application characterized by the presence of sizable data-flow computation and control. This model has been expressed using the basic model of computation supported by the Felix technology. An IP library has been created to favor design verification and re-use. The design has been imported into Felix to effectively explore the design space.

This work is supported by the European Commission under ESPRIT COSY EP25443. The first release of the COSY methodology and its evaluation with DVB designs is scheduled for Q1, 2000. The QSS technique is joint work with Jordi Cortadella at Universitat Politècnica de Catalunya, Spain. The authors wish to thank David Lahei, Ellen Sentovich, and Luciano Lavagno for their collaboration.

References

[1] F. Balarin et al, "Hardware-Software Co-Design of Embedded Systems: The Polis Approach", Kluwer, 1997

[2] G. Berry, "The Esterel v5 Language Primer, Version 5.10, release 2.0," Centre de Mathématiques Appliqées, Ecole des Mines and INRIA, France, 1998

[3] J-Y. Brunel et al., "COSY: a methodology for system design based on reusable hardware & software IP's," in: J-Y. Roger (ed.), *Technologies for the Information Society*, pp. 709-716, 1998

[4] J-Y. Brunel et al., "Communication Refinement in Video Systems On Chip," CODES'99, 1998

[5] W.A. Halang, A.D. Stoyenko, "Constructing predictable real time systems," Kluwer Academic Publishers, 1991

[6] G. Kahn, "The semantics of a simple language for parallel programming," in: J.L. Rosenfeld (ed.), Information Processing, North-Holland Publishing Co., pp. 471-474, 1974

[7] B. Kienhuis et al., "An approach for quantitative analysis of application-specific dataflow architectures," *ASAP'97*, Zurich, Zwitserland, July 14-16 1997, pp. 338-349

[8] L. Lavagno, E. Sentovich, "ECL: A Specification Environment for System-Level Design," *Submitted to DAC'99*, 1998

[9] E.A. Lee, D.G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Trans. on Computers*, January 1987

[10] G. Martin and B. Salefski, "Methodology and technology for design of communications and multimedia products via system-level IP integration," *DATE'98*, 1998

[11] T. Murata, "Petri nets: properties, analysis and applications," *Proc. of the IEEE*, April 1989

[12] M. Sgroi, "Quasi–static scheduling for free-choice Petri Nets," MS Thesis, UC Berkeley, 1998