

# YAPI: Application Modeling for Signal Processing Systems

E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf,  
J.-Y. Brunel, W.M. Kruijtzter, P. Lieveuse\*, K.A. Vissers

Philips Research, Prof. Holstlaan 4, 5656 AA, Eindhoven, The Netherlands  
\*Delft University of Technology, Mekelweg 4, 2628 CD, Delft, The Netherlands  
E-mail: Erwin.de.Kock@philips.com

**Abstract**— We present a programming interface called YAPI to model signal processing applications as process networks. The purpose of YAPI is to enable the reuse of signal processing applications and the mapping of signal processing applications onto heterogeneous systems that contain hardware and software components. To this end, YAPI separates the concerns of the application programmer, who determines the functionality of the system, and the system designer, who determines the implementation of the functionality. The proposed model of computation extends the existing model of Kahn process networks with channel selection to support non-deterministic events. We provide an efficient implementation of YAPI in the form of a C++ run-time library to execute the applications on a workstation. Subsequently, the applications are used by the system designer as input for mapping and performance analysis in the design of complex signal processing systems. We evaluate this methodology on the design of a digital video broadcast system-on-chip.

**Keywords**— Kahn process network, model of computation, application programming, signal processing, system-level design.

## I. INTRODUCTION

Modern signal processing systems such as digital televisions, set-top boxes, and mobile devices are multi-functional systems that support multiple standards. This calls for programmability. However, performance requirements and constraints on cost and power consumption still require that significant parts of these systems are implemented in dedicated hardware blocks. As a consequence, such systems have a heterogeneous architecture, i.e., they consist of programmable and dedicated components.

In our view the design of heterogeneous architectures should follow a general scheme, visualized by the Y-shape in Figure 1, hence, the name *Y-chart* [1] [2]. According to the Y-chart, the

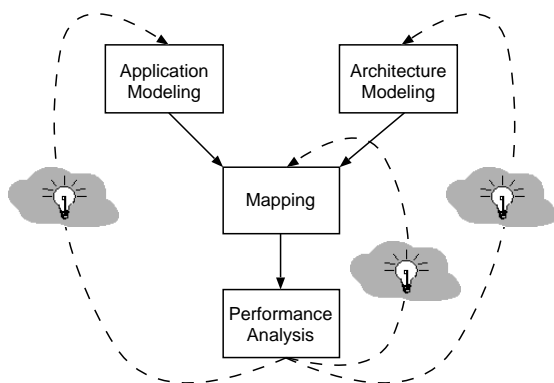


Fig. 1. The Y-chart: a general scheme for heterogeneous system design.

first step in the design includes the functional specification of a set of benchmark applications. Typically, a designer studies this set of applications, makes some initial calculations and proposes an architecture. Architectures are evaluated quantitatively by means of performance analysis. For this performance analysis, each application is mapped onto the architecture and the performance of each application-architecture-mapping combi-

nation is evaluated. The resulting performance numbers may inspire the architecture designer to improve the architecture. He may also decide to restructure the applications or to modify the mapping of the applications. These designer actions are denoted by the light bulbs in Figure 1. In this paper we focus on design technology for capturing the functional specifications of signal processing applications.

The outline of the paper is as follows. In Section II we present the requirements for capturing functional specifications. In Section III we motivate the model of computation underlying YAPI based on related work on the subject of application modeling. In Section IV we present the YAPI application programming interface. In Section V we illustrate the usage of YAPI with an example. In Section VI we discuss the mapping of YAPI onto target architectures. In Section VII we present the results concerning the modeling of a part of a digital video broadcast application. Finally, in Section VIII we draw some conclusions.

## II. REQUIREMENTS

Design technology for capturing the functional specifications of signal processing applications must satisfy a diverse set of requirements. The design technology must provide support for

- the composition of applications into larger applications to enable modular construction and reuse,
- the import of legacy code that is available in the form of sequential C-programs,
- the execution of applications to validate the functionality of the system by processing real data sets,
- the measuring of computation and communication requirements related to the processing of a data set, and
- the explicit expression of communication and parallelism to enable the mapping of applications onto heterogeneous architectures to provide a link to architecture design technologies.

## III. MOTIVATION AND RELATED WORK

Kahn process networks [3] is a model of computation that is often used for modeling signal processing applications. In this model, concurrent processes communicate through unidirectional first-in-first-out channels with unbounded capacity. Each of the processes performs sequential computation on its private state space. The computation actions are interleaved with communication actions that read data from input channels and write data to output channels. Read actions are blocking, i.e., a process that reads from an empty channel stalls until the channel has sufficient data to complete the read action. Write actions are non-blocking because the channels have unbounded capacity. A well-known property of a Kahn process network is that it is deterministic, i.e., the stream of data that travels along each chan-

nel is determined by the given input data; it does not depend on the order in which the processes are executed. For this reason, an application programmer can combine processes that represent signal processing functions into process networks without specifying their order of execution. Moreover, a system designer can exploit the concurrency between the processes by processing elements that operate in parallel.

In some implementations of Kahn process networks, the blocking read semantic incurs a considerable amount of context switching overhead. Dataflow process networks [4], which are a special case of Kahn process networks, avoid this overhead by transforming the processes into atomic actors that are fired when input data is available. Once an actor has been fired, it cannot stall. If it cannot complete the computation because it requires more input data, then it must save its internal state such that it can resume the computation on the next firing. The literature contains many references to variants of dataflow process networks such as synchronous or static dataflow [5], cyclostatic dataflow [6], and dynamic dataflow [7]. Many commercial vendors offer software packages for modeling signal processing systems based on dataflow process networks. Examples of such packages are SPW [8] and DSP Station [9].

We argue that dataflow process networks are less suited for modeling data-dependent applications because they place the burden of state saving on the application programmer rather than on the system designer. Explicit implementation of state saving by a dataflow programmer is a form of over-specification which can lead to unnecessary computation or communication. The reason for this is that the need for state saving and the implementation of state saving are design decisions. A system designer can avoid the need for state saving by avoiding resource sharing. If a system designer decides to apply resource sharing then an on-line resource scheduler with multi-tasking capabilities can provide automatic state saving. Only in case of resource sharing without multi-tasking capabilities there is a need for explicit state saving. For this reason we resort to the more general model of Kahn process networks which assumes implicit state saving, thereby leaving the use and the implementation of state saving as design decisions to a system designer.

A limitation of Kahn process networks is that they cannot model reactivity such as user interaction. This is caused by the fact that the absence and the occurrence of non-deterministic events cannot be made known to the processes. Control flow models such as finite state machines provide a solution for this problem by assuming a broadcast mechanism to communicate events in which each actor is sensitive to specific events. These models often contain a global notion of time such that time stamps can be associated with all events which is needed to process them in a correct order. This makes these models less suited for the implementation of computationally intensive applications because the amount of parallelism is limited. Furthermore, the underlying broadcast mechanism of these models is difficult to implement on parallel systems with distributed memory. Examples of software packages supporting control flow modeling are Statecharts [10], Esterel [11], and Polis [1].

The Ptolemy system [12] has been designed to support a heterogeneous mix of models of computation for co-simulation. It attempts to combine the semantics of control and data flow mod-

els at their interfaces [13]. Although this is feasible for functional simulation, we argue that this does not allow hardware software co-design because the models of computation are already tuned towards a target implementation. Another approach called Process Coordination Calculus [14] combines data-driven and event-driven processes in a single process network with stream-based, event-based, and register-based communication schemes. Here the terminology ‘process’ is confusing because the processes are in fact actors with firing rules. A specification consists of a process network and a set of scheduling constraints to ensure deterministic behavior.

To extend the deterministic model of Kahn process networks with non-deterministic events we pursue the approach of Martin [15], who has introduced a communication primitive known as the probe in combination with the model of Communicating Sequential Processes [16]. In this model concurrent processes communicate through unbuffered channels. As a result two communicating processes must complete their communication actions simultaneously. A probe action indicates whether the process on the opposite side of the channel is stalling because it has initiated a communication action that cannot be completed. Martin [17] demonstrates the use of probes to implement channel selection, i.e., selection of one channel out of a set of channels such that the next communication action on the selected channel can be completed. Since channel selection is performed at run-time it allows the modeling of non-deterministic events.

We generalize the notion of probes to buffered channels in order to extend the model of Kahn process networks with channel selection. Although channel selection can be implemented with probe actions, YAPI provides a more abstract operation because we argue that the implementation of channel selection is the concern of a system designer rather than of an application programmer. We hide the algorithm that selects a channel when there is more than one candidate from an application programmer, because the conditions that determine whether or not a channel is a candidate depend on design decisions such as the scheduling of shared resources, the computation delays, the communication delays, and many more. Since an application programmer does not know these design decisions, we do not allow an application programmer to control the channel selection algorithm. Therefore, we avoid probe-like constructs in YAPI that allow programmers to implement their own selection algorithm. We claim that the abstraction from these implementation details results in reusable applications that can be mapped onto different target architectures.

#### IV. YAPI DEFINITION

To describe the structure of a process network, we introduce the notions of process type set and data type. A process type set defines the set of process types. Each process type has a set of input ports and a set of output ports. With each port we associate a data type. A process network consists of a set of processes and a set of channels. A channel connects a process output port to a process input port of the same data type. Each port is connected to precisely one channel.

To describe the communication between the processes we provide three functions called *read*, *write*, and *select* that can be called from within a process. The informal meaning of these

functions is as follows. The read function consumes data from an input port and stores it in a local variable of the process. The write function copies the value of a local variable to an output port. The select function selects an input or output port that eventually will produce or consume data, respectively. To formalize the semantics of these functions we associate with each port at any time the number of tokens that it has transferred up to now and the number of tokens that it has committed to transfer up to now but that not have been transferred yet. To this end we introduce the following definitions.

**Definition 1.** Let  $p$  be a port. Then at any time

- $c(p)$  denotes the number of tokens transferred through  $p$ ,
- $n(p)$  denotes the number of tokens committed through  $p$ ,
- $m(p)$  denotes the port connected to  $p$  by a channel, and
- $v(p, k)$  denotes the value of the  $k$ -th token at  $p$ .  $\square$

Following the approach of Martin [18], the communication mechanism is based on the following assumptions. For all channels it must hold at any time that a write action is not blocked, the number of consumed tokens does not exceed the number of produced tokens, and the functionality is first-in-first-out. Formally, this is denoted as follows.

**Axiom 1.** Let  $(p, m(p))$  be a channel. Then at any time

- $n(p) = 0$ ,
- $c(m(p)) \leq c(p)$ , and
- $\forall_{0 \leq i < c(m(p))} (v(m(p), i) = v(p, i))$ .  $\square$

Under these assumptions we define the semantics of the read, write, and select functions using preconditions and postconditions as follows. Note that these functions stall when their postcondition cannot be satisfied. Furthermore, note that the write function is non-destructive which means that the variables of the producing process keep their values.

**Definition 2.** Let  $p$  be an input port of type  $t$ ,  $x$  an array of type  $t$ , and  $n$  a positive integer indicating a number of tokens. Then action  $read(p, x, n)$  is defined by precondition

- $c(p) = N$ , and postcondition
- $c(p) = N + n \wedge \forall_{0 \leq i < n} (x[i] = v(p, N + i))$ .  $\square$

**Definition 3.** Let  $p$  be an output port of type  $t$ ,  $x$  an array of type  $t$ , and  $n$  a positive integer indicating a number of tokens. Then action  $write(p, x, n)$  is defined by precondition

- $c(p) = N \wedge \forall_{0 \leq i < n} (x[i] = v(p, N + i))$ , and postcondition
- $c(p) = N + n \wedge \forall_{0 \leq i < n} (x[i] = v(p, N + i))$ .  $\square$

**Definition 4.** Let  $k$  be a positive integer,  $p_1$  up to  $p_k$  ports,  $n_1$  up to  $n_k$  positive integers indicating requests for numbers of tokens, and  $s$  a positive integer no larger than  $k$  indicating the index of the selected port. Then action  $s = select(p_1, n_1, \dots, p_k, n_k)$  is defined by precondition

- $\forall_{1 \leq i \leq k} (c(p_i) = N_i)$ , and postcondition
- $\forall_{1 \leq i \leq k} (c(p_i) = N_i) \wedge 1 \leq s \leq k \wedge \diamond (N_s + n_s \leq c(m(p_s)))$ .  $\square$

Following temporal logic theory, the symbol  $\diamond$  means eventually. Hence, the select function selects a port such that the current number of transferred tokens through this port plus the requested number of tokens is eventually smaller than or equal to the number of transferred tokens through the connected port. If we consider an input port  $p_s$ , then this port is a candidate for

selection if the corresponding output port  $m(p_s)$  will eventually produce enough tokens to complete a read action of  $n_s$  tokens. If we consider an output port  $p_s$ , then this port is a candidate for selection if the corresponding input port  $m(p_s)$  will eventually consume the tokens produced by a write action of  $n_s$  tokens. Note that the select function has no effect on the number of transferred tokens, i.e., it does not produce or consume data.

To describe the functionality of the processes we use a sequential programming language. We introduce an additional function called *execute* to abstract from the implementation of the functionality in the sequential programming language. To this end, the functionality between two communication actions has to be annotated with one or more execute actions. These execute actions annotate the computation requirements of the processes; they do not provide additional functionality.

## V. EXAMPLE

We illustrate the usage of YAPI with an example of a programmable filter that scales video lines. The purpose of the example is twofold. First we illustrate the combination of deterministic and non-deterministic communication. Second we illustrate the decoupling of the data types that are used for communication and computation. To this end we assume that the filter receives a stream of window widths from a window manager through input port  $p_1$  and a stream of pixels from a video source through input port  $p_2$ . The function of the filter is to scale the incoming video frames according to the incoming window widths and to transmit the resulting stream of pixels through output port  $p_3$  as outlined in the code fragment shown in Figure 2. The video source and the window manager are not syn-

---

```

for all frames  $f$  in sequence
begin
  if ( $select(p_1, 1, p_2, h \times w) == 1$ )
     $read(p_1, w', 1)$ ;
    for all lines  $l$  in frame  $f$ 
      begin
         $read(p_2, l, w)$ ;
        scale line  $l$  of width  $w$  to line  $l'$  of width  $w'$ ;
         $execute('scale')$ ;
         $write(p_3, l', w')$ ;
      end
    end
end

```

---

Fig. 2. Horizontal scaling of video lines.

chronized. We assume that the video source produces frames at a constant rate using write action  $write(p, f, h \times w)$ , where  $p$  is an output port and  $f$  is a video frame of  $h$  video lines that each contain  $w$  video pixels. The behavior of the window manager is unknown because it is controlled by the user. In order to cope with the non-deterministic behavior of the window manager we guard the read action of the window width in the filter with the select action  $select(p_1, 1, p_2, h \times w)$ . If the select action returns 1 then we initiate the read action  $read(p_1, w', 1)$  to obtain the new window width  $w'$ . If the select action returns 2 then we initiate the filtering of the next video frame. The resulting latency between input and output depends on the granularity of the filter. The filter shown in the example is line-based which means that it reads the video frame by  $h$  consecutive read actions

$read(p_2, l, w)$  where  $l$  represents a video line of  $w$  pixels thereby introducing a latency of one video line. Frame-based and pixel-based filters are also feasible. Note that in this scenario it is possible to scale two or more video frames to the same window width, but that it is not possible to change the width of a window in the middle of a video frame. In order to specify the latter, for instance to allow circular windows, the select action should be moved to the inner loop that iterates over the video lines.

## VI. MAPPING

The implementation of read, write, select, and execute actions is a concern of a system designer. Note that different read, write, select, and execute actions may be implemented in different ways, for instance, because some actions are executed in hardware and other actions are executed in software.

One of the design decisions is to determine the size of the fifos in order to obtain an implementation in finite memory. Deadlock can occur if they are too small, because the size of the fifos limits the set of reachable schedules. Going from unbounded to bounded fifos changes Axiom 1 such that for all channels it holds that at any time a write action and a read action are not blocked simultaneously, the number of produced tokens minus the number of consumed tokens is bounded by the size of the channel, and the functionality is first-in-first-out. Formally, this is denoted as follows.

**Axiom 2.** Let  $(p, m(p))$  be a channel of size  $s$ . Then at any time

- $(n(p) = 0) \vee (n(m(p)) = 0)$ ,
- $0 \leq c(p) - c(m(p)) \leq s$ , and
- $\forall_{0 \leq i < c(m(p))} (v(m(p), i) = v(p, i))$ .  $\square$

The read and write actions can be implemented such that the number of communicated tokens can exceed the size of the fifo. To this end, these actions have to be preempted when the number of committed tokens is not present or does not fit in the fifo.

Another design decision is the implementation of the notion of ‘eventually’ in the select function. For process networks that cannot be scheduled off-line, for instance due to data-dependent functionality, we have chosen to strengthen the expression  $\diamond(N_s + n_s \leq c(m(p_s)))$  in the postcondition of the select function to  $c(p_s) + n_s \leq c(m(p_s)) + n(m(p_s))$ . We obtain the number of committed tokens  $n(m(p_s))$  through the number of tokens of the read and write actions, i.e., the initiation of a read or write action of  $n$  tokens sets a commitment that decreases during transfer of the tokens. As a result the scheduling horizon is limited to one communication action, i.e., a select action takes one incomplete read or write action into account. Again this introduces deadlock if the size of the fifos is too small. Note that if the process network is a dataflow process network, then the firing rules can be implemented with select actions. In that case the read and write actions do not have to stall because the firing rules satisfy Axiom 2.

The above-mentioned design decisions have been implemented in a C++ run-time library that is used by application programmers to simulate the functionality of a process network on a workstation. Other YAPI implementations, in particular those proposed in COSY [19] and SPADE [20], target mixed hardware and software realizations in systems-on-chip. In the initial stage of the design process, these implementations are

abstract performance models to allow fast design space exploration. Subsequently, these performance models are refined into cycle-accurate models to allow final implementation.

## VII. RESULTS

We evaluated YAPI with an industrially relevant application called VIDEOTOP. The VIDEOTOP application describes part of the functionality of a digital video broadcast system. The system receives an MPEG2 transport stream, where the user selects the channels to be decoded. The associated video streams are then unscrambled, demultiplexed, and decoded. The user may also define post-processing operations on the decoded streams, such as zooming and composition. The top-level process network of

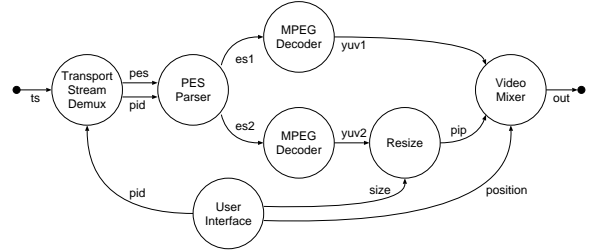


Fig. 3. The VIDEOTOP application.

the VIDEOTOP application is shown in Figure 3. The application consists of the following functions.

- An MPEG2 transport stream (ISO/IEC 13818-2) demultiplexer. This function extracts from an incoming transport stream (TS) those packetized elementary stream (PES) packets that correspond to the two packet identifiers (PID) selected by the user.
- An MPEG2 packetized elementary stream header parser. This function parses the incoming PES packets to collect elementary stream (ES) data per PID.
- An MPEG2 decoder. The H.262 compliant MPEG2 video bit-stream decoder decodes all video elementary streams up to main profile and high level (MP@HL).
- A video resizer. This function scales images horizontally and vertically with a zoom factor between 0.16 and 10 that is provided by the user. For the scaling we use horizontal and vertical sample rate converters that are implemented by polyphase filters with 6 taps and 64 phases.
- A video mixer. This function combines a number of arbitrary sized video images into a single new image. The positions and overlay priorities of the input images are controlled by the user.
- A user interface. This function provides the user with an interface to control the application. Upon changes of the user settings, it calculates and sends control data to several processes in the application.

The VIDEOTOP application has been simulated with the YAPI run-time environment to measure the requirements that the application imposes on the target architecture. We differentiate between two types of application requirements. The *communication requirement* is a measure for the amount of data that is transferred between processes. If two processes are executed on different processors, then there must be a communication link between the processors that has sufficient capacity to support this communication requirement. The *computation requirement* is a measure for the amount of computation that is performed to

execute a process. We need simulations to obtain these requirements because they may depend on the input data. The communication requirements that were measured for the VIDEOTOP application are listed in Table I. The table shows the measure-

TABLE I  
COMMUNICATION REQUIREMENTS OF VIDEOTOP.

stream	token count	byte count	bytes/s
ts	3,230,720	12,922,880	5,048,000
pes	7,524,002	30,096,008	11,756,253
es1	5,100,612	5,100,612	1,992,427
es2	2,380,224	2,380,224	929,775
yuv1	53,084,160	106,168,320	41,472,000
yuv2	53,084,160	106,168,320	41,472,000
pip	13,271,040	26,542,080	10,368,000
out	53,084,160	106,168,320	41,472,000
total			154,510,465

ments for a simulation of 64 video frames, which represents 2.56 seconds of video at a frame rate of 25 Hz. In this application the YUV video streams after the MPEG decoders require the highest data rates with about 40 MB/s per stream. One of the video streams is scaled down by a factor of two in both directions, and thus takes four times less bandwidth, i.e., 10 MB/s. From the table we conclude that for the given input data a communication bandwidth of at least 150 MB/s is needed. Hence, we might choose an architecture where all processes are implemented on separate processors that communicate via a single bus of 32 bits running at 100 MHz. This initial architecture can then be used as input for a more detailed mapping and performance analysis. The scheduling of the bus requires further analysis because the PES and ES streams have a variable data rate which is caused by the differences in compression of intra, predicted, and bidirectionally predicted frames. The variable data rate of the ES streams is shown in Figure 4. The above-mentioned simulation

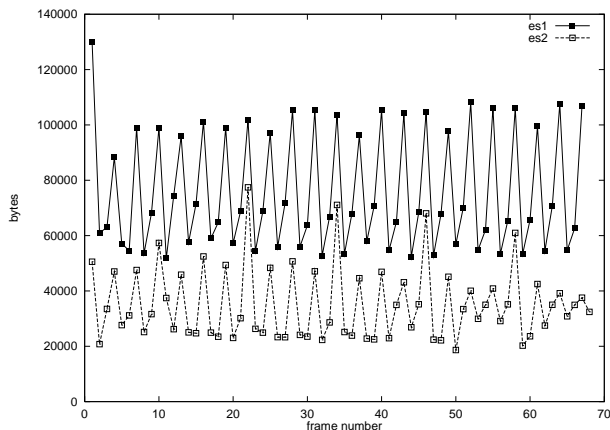


Fig. 4. Communication requirements of the elementary streams per frame.

requires approximately 14 minutes on an HP735 workstation, which is 325 times slower than real-time.

## VIII. CONCLUSION

We have presented an application programming interface called YAPI for capturing functional specifications of signal processing applications as process networks that supports deterministic and non-deterministic communication. The concept of process networks allows composition which enables modular con-

struction and reuse. We provide an implementation of YAPI in C++ such that legacy C-code can be imported easily and the applications can be compiled, executed, and analyzed on a workstation. This implementation has a small code base, is easy to use, and has a small run-time overhead. The underlying model of computation has been chosen such that the programmer can easily specify data dependencies while making communication and parallelism explicit. However, the programmer cannot control design decisions such as the scheduling of communication and computation. This key issue allows the mapping of YAPI applications onto a variety of heterogeneous target architectures.

## ACKNOWLEDGEMENTS

Part of this work has been sponsored by the European Commission under ESPRIT COSY EP25443.

## REFERENCES

- [1] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, and K. Suzuki, *Hardware-software co-design of embedded systems: the Polis approach*, Kluwer, 1997.
- [2] B. Kienhuis, E. Depretre, K. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 1997, pp. 338–349.
- [3] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing*, J.L. Rosenfeld, Ed. North-Holland Publishing Co., 1974.
- [4] E.A. Lee and T.M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, pp. 773–801, 1995.
- [5] E.A. Lee and D.G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, 1987.
- [6] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Static scheduling of multi-rate cyclo-static DSP applications," in *Proceedings Workshop on VLSI Signal Processing*, 1994, pp. 137–146.
- [7] J.T. Buck, "Static scheduling of code generation from dynamic dataflow graphs with integer valued control signals," in *Asilomar Conference on Signals, Systems and Computers*, 1994, pp. 508–513.
- [8] Cadence Design Systems, "SPW user's manual," 919 E. Hillsdale Boulevard, Foster City, CA 94404, USA.
- [9] Mentor Graphics, "DSP station user's manual," 1001 Ridder Park Drive, San Jose, CA 95131, USA.
- [10] M. von der Beeck, "A comparison of statecharts variants," in *Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems*, 1994, pp. 128–148.
- [11] F. Boussinot and R. de Simone, "The Esterel language," *Proceedings of the IEEE*, vol. 79, pp. 1293–1304, 1991.
- [12] J. Buck, S. Ha., E.A. Lee, and D.G. Messerschmitt, "A platform for heterogeneous simulation and prototyping," in *Proceedings of the 1991 European Simulation Conference*, 1991.
- [13] W. Chang, S. Ha, and E.A. Lee, "Heterogeneous simulation - mixing discrete-event models with dataflow," *Journal of VLSI Processing*, vol. 15, pp. 127–144, 1997.
- [14] T. Grötter, R. Schoenen, and H. Meyr, "PCC: A modeling technique for mixed control/data flow systems," in *European Design & Test Conference*, 1997, pp. 482–486.
- [15] A.J. Martin, "The probe: An addition to communication primitives," *Information Processing Letters*, vol. 20, pp. 125–130, 1985.
- [16] C.A.R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, pp. 666–677, 1978.
- [17] A.J. Martin, "Programming in VLSI: From communicating processes to delay-insensitive circuits," in *Developments in Concurrency and Communication*, C.A.R. Hoare, Ed. Addison-Wesley, 1990.
- [18] A.J. Martin, "An axiomatic definition of synchronization primitives," *Acta Informatica*, vol. 16, pp. 219–235, 1981.
- [19] J.-Y. Brunel, E.A. de Kock, W.M. Kruijtzter, H.J.H.N. Kenter, and W.J.M. Smits, "Communication refinement in video systems on chip," in *International Workshop on Hardware/Software Codesign*, 1999, pp. 142–146.
- [20] P. van der Wolf, P. Lieverse, M. Goel, D. La Hei, and K. Vissers, "An MPEG-2 decoder case study as a driver for a system level design methodology," in *International Workshop on Hardware/Software Codesign*, 1999, pp. 33–37.