1 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 [2] [1]. According to the Y-chart, the 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. To this end, each application is mapped onto the architecture and the performance of each application-architecture-mapping combination 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 2 we present the requirements for capturing functional specifications. In Section 3 we motivate the model of computation underlying YAPI based on related work on the subject of application modeling. In Section 4 we present the YAPI application programming interface. In Section 5 we illustrate the use of YAPI with an example. In Section 6 we discuss the mapping of YAPI onto target architectures. In Section 7 we present the results concerning the modeling of a part of a digital video broadcast application. Finally, in Section 8 we draw some conclusions.

2 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 (a) composing applications into larger applications to enable modular construction and reuse, (b) importing legacy code that is available in the form of C code, (c) executing applications to validate the functionality of the system by processing real data sets, (d) measuring computation and communication requirements related to the processing of a data set to determine the workload imposed on target architectures, and (e) explicit expression of communication and parallelism to enable the mapping of applications onto target architectures to provide a link to architecture design technologies.

3 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 channel 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 using 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], cyclo-
static dataflow [6], and dynamic dataflow [7]. Many commercial ven-
dors 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 mod-
eling data-dependent applications because they place the burden of
state saving on the application programmer rather than on the sys-
tem designer. Explicit implementation of state saving by a dataflow
programmer is a form of over-specification which can lead to unnec-
essary 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 sav-
ing, 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
reactiveness 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 fi-
ite 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 inten-
sive applications because the amount of parallelism is limited. Fur-
thermore, 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 heteroge-
nous mix of models of computation for co-simulation. It attempts
to combine the semantics of control and data flow models at their
interfaces [13]. Although this is feasible for functional simulation,
we argue that this does not allow hardware software co-design be-
cause the models of computation are already tuned towards a tar-
get 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 un-
buffered channels. As a result two communicating processes must
complete their communication actions simultaneously. A probe ac-
tion indicates whether the process on the opposite side of the channel
is stalled because it has initiated a communication action that cannot
be completed. Martin [17] demonstrates the use of probes to im-
plement 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 selec-
tion. 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 sys-
tem designer rather than of an application programmer. We hide the
algorithm that selects a channel when there is more than one can-
didate 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 computa-
tion 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 se-
lection 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 re-
results in reusable applications that can be mapped onto different target
architectures.

4 YAPI Definition. To describe the structure of a process net-
work, 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 fol-
lo ws. 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 se-
lects 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 \( p \),
- \( m(p) \) 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 \( p \).

Following the approach of Martin [18], the communication mecha-
nism 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 con-
sumed tokens does not exceed the number of produced tokens, and
the functionality is first-in-first-out.

Axiom 1. Let \( (p,m(p)) \) be a channel. Then at any time
- \( m(p) = 0 \),
- \( c(m(p)) \leq c(p) \), and
- \( v_{0 \leq i < c(p)}(v(m(p),i)) = v(p,i) \).

Under these assumptions we define the semantics of the read, write,
and select functions using preconditions and postconditions as fol-
lo ws. Note that these functions stall when their postcondition can-
not 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(\( p,t,n \)) is defined by precondition

\[ \cdots \]
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 \( \text{write}(x, n) \) is defined by precondition
\[
c(p) = N + n \land \forall_{0 \leq i < c}(x[i] = v(p, N + i))
\]
and postcondition
\[
c(p) = N + n + \forall_{0 \leq i < c}(x[i] = v(p, N + i))
\]

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 \( s = \text{select}(p_1, p_2, \ldots, p_k, n_1) \) is defined by precondition
\[
\forall_{1 \leq i \leq k}(c(p_i) = N_i), \text{ and postcondition}
\]
\[
\forall_{1 \leq i \leq k}(c(p_i) = N_i) \land 1 \leq s \leq k \land \bigcirc(N_i + n_s \leq c(m(p_i)))
\]
Following temporal logic theory, the symbol \( \bigcirc \) 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 input port \( m(p_s) \) eventually produces 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 \( \text{execute} \) 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.

Example. We illustrate the use of \textsc{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 synchronized. We assume that the video source produces frames at a constant rate using write action \( \text{write}(p, f, h \times w) \), 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 \( \text{select}(p_1, p_2, h \times w) \). If the select action returns 1 then we initiate the read action \( \text{read}(p_1, w', 1) \) 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 \( \text{read}(p_2, i, w) \) where \( i \) 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 window width of a window in the middle of a video frame. In order to specify the latter, for instance to allow non-rectangular windows, the select action should be moved to the inner loop that iterates over the video lines.

6 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
\[
(n(p) - 0) \lor (n(m(p)) = 0),
\]
\[
0 \leq c(p) - c(m(p)) \leq s,
\]
\[
\forall_{0 \leq i \leq c(m(p))}(v(m(p), i) = v(p, i)).
\]

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 \( \bigcirc(N_i + n_s \leq c(m(p_s))) \) 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.

We have implemented a C++ run-time library, based on the above-mentioned design decisions, that is used by application programmers to simulate the functionality of a process network on a workstation. Other \textsc{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.

7 Results. We evaluated \textsc{YAPI} with an industrially relevant application called \textsc{videotop}. The \textsc{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 the VIDEOTOP application consists of the functions shown in Figure 3.

![Figure 3. The VIDEOTOP application.](image)

- A transport stream demultiplexer extracts from an incoming transport stream (TS) those packetized elementary streams (PES) that correspond to two packet identifiers (PID) selected by the user.
- A packetized elementary stream header parser parses the incoming PES packets to collect elementary stream (ES) data per PID.
- An H.262 compliant video bitstream decoder decodes all video elementary streams up to main profile and high level.
- A video resizer 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 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 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 runtime environment to measure the workload that the application imposes on a target architecture. We differentiate between two types of 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 1. The table shows the measurements 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 requires 14 minutes on an HP735 workstation, which is 325 times slower than real-time.

![Figure 4. Communication requirements of the elementary streams per frame.](image)

8 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 construction 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 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.