Reactive Process Networks*

Marc Geilen and Twan Basten
Eindhoven University of Technology
Department of Electrical Engineering
P.O.Box 513, 5600 MB, Eindhoven, The Netherlands
M.C.W.Geilen@tue.nl, A.A.Basten@tue.nl

ABSTRACT

Data flow process networks are a good model of computation for streaming multimedia applications incorporating audio, video and/or graphics streams. Process networks are concurrent processes communicating streams of data through FIFO channels. They can be executed efficiently and determinately on multiprocessor platforms. However, such stream processing applications are becoming more dynamic, often requiring run-time reconfigurations. Moreover, stream processing is not always an application on its own, but may be a component of a larger application. This application, e.g. a game application, may be control oriented and event driven; events may interact with the streaming component and (re)configure it. In order to capture the interaction between reactive and streaming components as well as reconfiguration in dynamic stream processing, we introduce in this paper a formal, operational and compositional semantics of so-called reactive process networks. This operational semantics can serve as the basis for programming models that allow the programming of streaming components interacting with reactive system components and their reconfigurations. It also supports the construction of analysis and synthesis tools for dynamic streaming multimedia applications. It allows the integration of reactive behaviour in process networks as general as Kahn process networks, but it is also suitable for more restricted and efficient classes of process networks.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*

General Terms

Languages, Theory

*This work is supported by the IST-2000-30026 project, Ozone.

EMSOFT'04, September 27–29, 2004, Pisa, Italy.

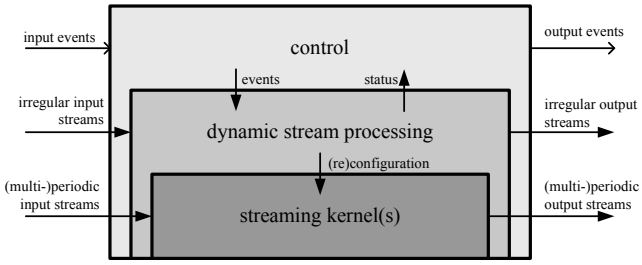

Figure 1: Embedding of types of streams

Keywords

process networks, operational semantics, multiprocessor systems, media processing, signal processing, reactive systems

1. INTRODUCTION

Multimedia applications work with information streams such as audio, video or graphics. With modern applications, these streams and their encodings can be very dynamic. Smart compression, encoding and scalability features make these streams less regular than they used to be. Streams are typically parts of larger applications. Other parts of these applications tend to be event-driven and interact with the streaming components. Modern (embedded) multimedia applications can often be seen as instances of the structure depicted in Figure 1. At the heart of the application, computationally intensive data operations have to be performed in streams of for instance pixels, audio samples or video frames. Input and output of these processes are highly regular patterns of data. These data processing activities can often be statically analysed and scheduled on efficient processing units. At a higher level, modern multimedia streams show a lot of dynamism. Object-based video (de)coders for instance work with dynamic numbers of objects that enter or leave a scene. Decoding of the individual objects themselves uses the static data processing functions, but they may need to be added or removed dynamically. These dynamic streams still compute functions and processing is determinate, i.e. the functional result is independent of the order in which operations are executed. In turn, the processing of these dynamic data streams is governed by control oriented components. This may for instance be used to convey user interactions to the streaming application or to respond to changing network conditions.

In our view, the three levels of an application require typical modelling and implementation techniques. A good

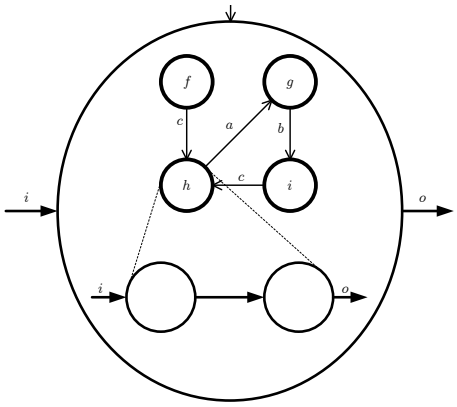

Figure 2: Automaton with process networks

candidate for instance to describe static computation kernels is Synchronous Data Flow (SDF) [13]. SDF graphs consist of actors describing data transformations with fixed data consumption and production rates, allowing them to be executed according to a periodic schedule, which can be determined at design time. Dynamic stream processing can be described using Kahn Process Networks (KPNs) [9]. KPNs are also networks of processes operating on streams of data. They are however capable of showing data dependent behaviour and dynamic changes in their processing. Yet, they are still determinate and can be executed fully asynchronous. KPNs allow for dynamic network reconfigurations (even if many of the current implementations of KPN do not). To specify the control dominated parts of an application, there are many techniques, such as state machines and event-driven software.

It is the intention of this paper to define a model and formal operational semantics that allows for an integrated description of an application consisting of these three levels of computation. A unified model is presented for streaming and control, that is also hierarchical and fully compositional. This model can be used as the basis for constructing analysis or synthesis tools or defining programming systems for realising streaming media applications on programming platforms with reconfigurations and control. Moreover, we feel that defining a formal semantics can be very important for understanding the subtle details of the models and for verifying what we believe to know about these models. For traditional KPNs for instance, formulating a formal operational semantics of their execution allowed us to discover that a popular execution model could fail under specific conditions [6].

Figure 2 illustrates what we intuitively try to achieve. A process network is a component with stream input(s) (i in Figure 2), stream output(s) (o) and event input(s) (e).

At any point in time, the network operates in a mode that implements a particular streaming function, for instance mode h in Figure 2, implemented as the network drawn below it. At some later time, because of the occurrence of some event a, the function of the network needs to change to a mode g having a different streaming function. One could think for instance of a video system where a user changes settings or turns on or off special features. One could view this as a (finite) state machine where in every state, the network implements a particular function and external events force the state machine to move from one state to another. In every state, a particular process network performs operations on data streams. Since process networks can be hierarchical entities, we would like such a construct to be compositional and applicable at different levels of a network hierarchy.

The remainder of this paper is structured as follows. The following section discusses related work. Section 3 discusses the principles behind our definition of Reactive Process Networks (RPN). An example is introduced in Section 4. This is followed by the formal definition of Reactive Process Networks and their formal operational semantics in Section 5. Some implementation considerations for realising RPNs are the topic of Section 6, followed by conclusions and suggestions for future work in Section 7.

2. RELATED WORK

Process networks and data flow models are very popular for specification, analysis and synthesis of high performance signal processing and streaming multimedia applications [13]. Advanced analysis and automatic synthesis techniques exist for a class of dataflow systems called synchronous dataflow (SDF) and extensions of SDF that preserve the static analysis options, such as cyclo-static dataflow [3]. More expressive classes of process networks or dataflow systems exist, such as dynamic dataflow systems or Kahn Process Networks (KPNs). Their expressiveness disables many of the advanced static and compile time analysis methods and run-time scheduling and resource management and arbitration is required. Many of the existing run-time environments for such systems are commonly used as simulation frameworks rather than actual implementations [12, 8, 10].

The desire to express non-deterministic behaviour and event-based communication has lead to additions to pure data flow models. Examples are the probes of [15] and [10]. [15] describes an extension of the dataflow model with the ability to probe a channel for the presence or absence of data. While this enhances the expressiveness of the model, it destroys the property of determinacy, of independence of any concrete schedule. This probe construct inspired the designers of Yapi [10] to introduce the select statement, having the same disadvantage. In Ptolemy [12, 8], a framework is defined to connect multiple models of computation, including data flow and event-based ones. The combination of the reactive and process network domains however, induce too much synchronisation overhead to be used for implementation.

Many of the combinations of data flow and reactive behaviour are based on a combination of the Synchronous Data Flow model together with some form of reactive behaviour [19, 11, 18, 7]. The use of an analysable model such as SDF is natural because it allows for a predictable and determinate combination. The reactive part is frequently specified using (hierarchical) state machines.

[11] describes a combination of hierarchical state machines and SDF models, where a complete iteration of the SDF is taken as an action of the state machine. A state machine inside an SDF is required to adhere to the SDF firing characteristics. FunState [18] defines a model, described as 'functions driven by state machines', which deliberately tries to separate data flow from control. *charts [7] separates hierarchical finite state machine models from concurrency models. In the data flow domain, a combination of Hierarchical

Finite State Machines with Synchronous Data Flow is presented.

[2] describes parameterisable SDF models, allowing dynamic reconfiguration during runtime. Processes can change their dataflow behaviour depending on parameter settings. In a given SDF configuration, actor executions are characterised by iterations, that fire subprocesses in a particular order that returns the internal buffer states in their original configuration. Such a process can (only) be reconfigured between such iterations and if the dataflow behaviour has changed, a new schedule is determined (at run-time).

Most similar to our approach is a very recent paper [16] of Neuendorffer and Lee. They focus on reconfiguration as a particular kind of event handling. They define *quiescent states* as the states where reconfigurations are allowed. These quiescent states are strongly related to our maximal streaming transactions. They also propose, similar to this paper, to use FIFO (First In First Out) channel communication also for events or parameters and to divide input ports in streaming input ports and parameter input ports. In contrast with [16], we consider more general event handling and reconfiguration than changing parameters. It also focusses primarily on analysability and schedulability of reconfigurable SDF graphs, whereas we start from more general KPNs.

What makes this paper different from other work, is that we look at dataflow systems in the style of KPNs (hence, more general than SDF which is often used), because we feel that that is required for many modern multimedia applications. We furthermore consider the possibility of dynamically changing the structure of the process network as opposed to reconfiguration by only changing parameters. Instead of combining a data flow graph with a state machine, we seek to define a single, unified model, that is also fully compositional. Moreover, we give a formal semantics which enables rigourous analysis of our model and systems specified using this model.

3. DESIGN PRINCIPLES

In this section, we discuss the basic ideas that have had a major impact on the design of our model of Reactive Process Networks.

3.1 Streams, Events and Time

Streaming applications represent functions or data transformations. Presented with input (strings of tokens offered to input ports) they produce output (strings of tokens produced at output ports). There is typically no inherent notion of time, except for the ordering of tokens in the individual data streams. There is no relation in time between tokens in different streams, except for causal orderings implicitly defined by the processes that operate on the tokens. These process networks are determinate, i.e. the order in which processes execute is irrelevant for the functional result. The reaction of a process network is conceptually immediate (the output is determined as soon as the input is known). However, it is understood that the actual production of the output introduces a certain latency in the reaction. This latency is not inherent in the functional specification, but merely a byproduct of the computation process. It may be subject to constraints, such as a maximum latency. Time is sometimes implicitly present in the intention of steams. A stream may carry for instance, a sequence of samples of an audio signal that are 1/44100th of a second apart, or video frames of which there are 25 or 30 in every second. Such streams are called *periodic*. Time does play a role of course in realisations. Then, time-related notions such as throughput, latency and jitter are important.

Events, in contrast with streams, rely heavily on a notion of time. An event is unpredictable and whether it will arrive and when it will arrive is significant, but unknown in advance. In many event-based models, the synchrony hypothesis applies, which states that the response to an event can be completed before the following event arrives or is taken into account. A classical model for event based systems are labelled transition systems. Prominent characteristics are non-determinism and a total ordering of events. This total ordering of all events introduces a global notion of qualitative time and when mixed with explicit time related events such as clock ticks or delays, even a quantitative notion of time. Timing constraints on events in realisations are typically response time constraints, i.e. the duration between the arrival of an event and the manifestation of the associated effect.

The integration of events in stream based execution, implicitly introduces a sense of time in the stream processing that is not there originally. This has had a major impact on the synchronisation constraints that govern the interaction of events and streams.

3.2 Semantics of Process Networks

3.2.1 Denotational and operational semantics

There are two common ways to describe a network of dataflow processes, namely denotational approaches and operational approaches. Every process (or 'actor' in some models) realises a certain function, consuming input tokens and computing a functional result which is then written to its outputs. The denotational interpretation of the network as a whole can then be derived from the composition of these individual functions. If the network is recursive, i.e. if it contains cycles, then the composition can be expressed using a (set of) fixed-point equation(s) over these functions [9]. Alternatively, a network can be characterised operationally. Processes read tokens from channels or write tokens to channels and perform computations in the mean time. The channels that connect processes store tokens that are in transit from one process to another. Denotational semantics is often preferred to capture the intended functionality of a process network or to define the functional semantics of a system or programming language implementing process networks, without specifying unnecessary implementation details. The operational semantics on the other hand allows reasoning about implementation details, such as artificial deadlocks [6] or required buffer capacities [3, 1]. For KPNs, the relationship between both styles of semantics is known as the Kahn Principle. Kahn predicted [9], and others later formally proved [5, 14], that both semantics define the same behaviour. The streams of tokens that are incrementally produced on the channels in the operational model realise the desired solution to the networks fixed-point equations.

3.2.2 Labelled Transition Systems

To formalise our reactive process networks, we use an operational semantics in the form of *labelled transition systems* (LTSs). A labelled transition system does not have, in gen-

eral, a functional representation that matches the domain of streams. Streams, on the other hand, can be modelled by (a special class of) labelled transition systems, albeit at a somewhat lower level of abstraction, introducing possibly irrelevant details [14]. Causality information is explicit in a dataflow representation, but becomes implicit in an LTS representation. However, the discussion in Section 3.1 makes clear that dealing explicitly with the timing or order of input and output actions of the streams is necessary to describe interactions between streams and events, which is possible in an LTS-based approach.

3.2.3 Model of Streaming Applications

In this paper, we use Kahn Process Networks as the general model of computation for stream based applications, but it encapsulates also more restricted models such as SDF for periodic streams. Traditionally, the behaviour of a KPN is described by a function that maps a complete input history (all input up to some point in time) to all the corresponding output. This formulation allows a KPN to base output not only on the current input, but also on the history of inputs, i.e. to exploit this history as some notion of memory or 'state'. This is in contrast with SDF, which does not have this kind of memory and for every actor firing, computed output tokens are a function of the consumed input tokens. In an operational semantics of KPNs [5, 14], this state is made explicit and also the reading of input and the production of the corresponding output are decoupled.

3.2.4 Streaming and Time

We have further inspiration from comparing our idea of transition systems of streaming process networks to models of timed reactive systems. The passing of time is analogous to the flow of the stream, with the difference that time is usually assumed to pass synchronously in all components and the flow of streams allows some local variation as long as causality constraints are respected. Discrete changes are interleaved with stable periods of streaming. Conceptually, a discrete change occurs at a well defined point within the stream. Because of pipelining implementation of the stream processing however, there is not necessarily a point in time where the change can be applied instantaneously to the whole network. Hence, an important aspect of dealing with streaming computation and events will be to coordinate their execution to implement a smooth transition.

3.2.5 Synchronising Events with Streams

There is a trade-off between predictability and synchronisation overhead. Predictability of processing (non deterministic) events is improved by added control over the moment when and the way in which the event is processed relative to the streaming activities. We will see that increased predictability requires more synchronisation between processes and hence additional overhead. Such overhead is undesirable, especially if events occur only sporadically.

3.2.6 Communicating Events

Processes or actors in data flow graphs communicate via FIFO channels. We want to add communication of events and we have to decide what communication mechanism is used for events. It is often the case that what is perceived by a lower level process as an event, is considered to be part of streaming by higher level processes. For instance, a video

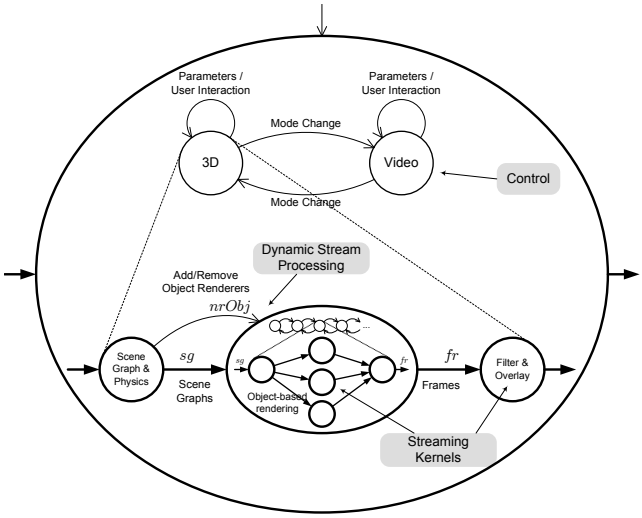

Figure 3: An interactive 3D game

decoder is decoding a stream of video frames and header information for every frame is an integral part of the data stream. For the lower level frame decoder processes. The frame header information is seen as an event that initialises the component to deal with the specific parameters of the following frame. For this reason, we use the same infrastructure for communicating streams also for events. For the sending process, there is no difference at all; at the receiving side, we distinguish *stream input* ports and *event input* ports. The former are used in ordinary streaming activity; tokens arriving on the latter will trigger discrete events.

4. AN EXAMPLE

An example of the type of application we are considering is shown in Figure 3. It depicts an imaginary game, which includes modes of 3-dimensional game play with streaming video based modes and (conceptually) how it is formalised. The rendering pipeline, used in the 3D mode, is a dynamic streaming application. Characters or objects may enter or leave the scene because of player interaction, rendering parameters may be adapted to achieve the required frame rates. Overlayed graphics (for instance text or scores) may change. This happens under control of the event-driven game control logic. At the core of the application, the streaming kernels, a lot of intensive pixel based operations are required to perform the various texture mapping or video filtering operations. Special hardware or processors may be available to execute these operations, which can be scheduled off-line, very efficiently.

Figure 3 shows (part of) an RPN model of our game. The model is organised around the two main modes (3D graphics, video). In these modes, the game dynamics and mode changes are influenced or initiated by user interaction, game play and performance feedback. This is the control oriented part of the game depicted as the automaton at the top of Figure 3. The self-loops on these states denote changes where the streaming network essentially stays the same, but its parameters may be changed. In the 3D graphics mode (enlarged at the bottom of the figure), a scene graph, describing all entities and their positions in 3D space, is rendered to a 2-dimensional view on the scene on some output device.

The first process communicates the scene graphs with objects to the rendering component. The rendering component transforms the scene graphs into 2-dimensional frames. The last process adds 2-dimensional video processing such as filtering, overlays, and so forth. The output is shown by the display device.

Notice that the game as a whole has different modes of streaming (3D graphics, video), but similarly, components in the stream processing part have different modes or states of streaming execution. The object renderer for instance can be reconfigured to different modes depending on the number of objects that need to be rendered (using the event channel *nrObj* controlled by the scene graph process). This suggests the need for a hierarchical, compositional approach to combining state/event based models with streaming and data flow based models as realised by our model.

5. REACTIVE PROCESS NETWORKS AND THEIR SEMANTICS

In this section, we formally define Reactive Process Networks and construct their operational semantics.

5.1 Preliminary Definitions

We start with some preliminary definitions and notations. We assume a universal, countable, set *Chans* of channels and for every channel $c \in Chans$ a corresponding countable channel alphabet Σ_c. From channels and their alphabets, we build a universal set of *actions* $Act = \{c?a, c!a \mid c \in Chans, a \in \Sigma_c\}$, consisting of *input actions* ($c?a$) and *output actions* ($c!a$). We use Σ to denote the union of all channel alphabets, and A^* (A^∞) to denote the set of all finite (and infinite) strings over alphabet A. $\sqsubseteq$ denotes the prefix relation on strings (a complete partial order). If σ and τ are strings, $\sigma \cdot \tau$ denotes the usual concatenation of the strings and $\sigma - \tau$ the string σ after removing the common prefix with τ. In particular, if $\sigma = \tau\upsilon$ then $\sigma - \tau = \upsilon$.

DEFINITION 5.1. (HISTORY) *A* history h *of a set* $C \subseteq Chans$ *of channels is a mapping from channels* $c \in C$ *to strings over* Σ_c. *The set of all histories of* C *is denoted as* $H(C)$.

A history can be used to capture the data communicated between two processes at some point in time or the data that is still in transit. A history of the channels *sg* and *fr* from our example, could for instance be the function $\{(sg, sg_1 \cdot sg_2); (fr, fr_1 \cdot fr_2 \cdot fr_3)\}$, which assigns to channel *sg* the sequence $sg_1 \cdot sg_2$ of scene graph data tokens and to channel *fr* the sequence $fr_1 \cdot fr_2 \cdot fr_3$ of three video frames.

If h_1 is a history of C_1 and h_2 is a history of C_2, we write $h_1 \sqsubseteq h_2$ if $C_1 \subseteq C_2$ and for every $c \in C_1$, $h_1(c) \sqsubseteq h_2(c)$. If $h_1, h_2 \in H(C)$, then the concatenation $h_1 \cdot h_2$ is the history such that $h_1 \cdot h_2(c) = h_1(c) \cdot h_2(c)$ for all $c \in C$. If $h_1 \sqsubseteq h_2$, then $h_2 - h_1$ denotes the history h_3 such that $h_1 \cdot h_3 = h_2$.

We are going to give an operational semantics to RPNs in the form of a labelled transition system (LTS). We use, more specifically, an LTS with an initial state, with designated streaming or event input actions and output actions in the form of reads and writes of tokens on channels, as well as internal actions.

DEFINITION 5.2. (LTS) *A* labelled transition system *is a tuple* $(S, \hat{s}, I, O, A, \longrightarrow)$ *consisting of a (countable) set* S *of states, an initial state* $\hat{s} \in S$, *a set* $I \subseteq Chans$ *of input channels, a set* $O \subseteq Chans$ *(disjoint from* I*) of output channels, a set* $A \subseteq Act^*$ *of sequences of actions consisting of input actions* $\{c?a \mid c \in I, a \in \Sigma_c\} \subseteq Act$, *output actions* $\{c!a \mid c \in O, a \in \Sigma_c\} \subseteq Act$ *and (possibly) internal actions (all other actions), and a labelled transition relation* $\longrightarrow \subseteq S \times A \times S$.

Note that we allow labels of the LTS to be sequences of read and write actions, because we will need to group such sequences into single, atomic transitions for our semantics. We write $s_1 \xrightarrow{\alpha} s_2$ if $(s_1, \alpha, s_2) \in \longrightarrow$, which denotes that the LTS in state s_1 can perform action(s) α which brings it to state s_2. Moreover, we write $s_1 \xrightarrow{\alpha}$ if there is some $s_2 \in S$ such that $s_1 \xrightarrow{\alpha} s_2$. With a write operation to an output channel, the token on the channel is determined by the LTS. With a read operation on an input channel, the token that appears on the channel is determined by the environment of the LTS. Therefore, a read operation of the RPN needs to be modelled with a set of input actions that provides a transition for every possible token of the alphabet.

DEFINITION 5.3. (EXECUTION) *An* execution σ *of a transition system is a sequence* $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ *of states* $s_i \in S$ *and actions* $\alpha_i \in Act$, *such that* $s_i \xrightarrow{\alpha_i} s_{i+1}$ *for all* $i \geq 0$ *(up to the length of the execution).*

If σ is such an execution, then we use $|\sigma| \in \mathbb{N} \cup \{\infty\}$ to denote the length of the execution. $|\sigma| = \infty$ if σ is infinite and $|\sigma| = n$ if $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n$. For $k \leq |\sigma|$, we use σ^k to denote the prefix of σ of length k.

From a given execution σ with actions $\alpha = \alpha_0 \cdot \alpha_1 \cdot \ldots$, we extract the consumed input and the produced output on a set $C \subseteq Chans$ of channels as follows.

- For a channel $c \in C$, $\alpha?c$ is a (finite or infinite) string over Σ_c that results from projecting α onto *read* actions on c;
- similarly, $\alpha!c$ is a (finite or infinite) string over Σ_c that results from projecting α onto *write* actions on c;
- finally, input history $\alpha?C = \{(c, \alpha?c) \mid c \in C\}$ and output history $\alpha!C = \{(c, \alpha!c) \mid c \in C\}$.

Furthermore, we use the same notation for executions: $\sigma?c = \alpha?c$, $\sigma!c = \alpha!c$, $\sigma?C = \alpha?C$ and $\sigma!C = \alpha!C$. Thus $\sigma?I$ denotes the input consumed by the network in execution σ and $\sigma!O$ denotes the output produced by the network. The *I/O-history* $h(\sigma)$ of an execution σ is $\sigma?I \cup \sigma!O$; if α is the string of actions executed in execution σ, $h(\alpha) = h(\sigma)$. To reason about the input *offered* to the network (consumed or not consumed), we say that σ is an execution with input $i : I \to \Sigma^\infty$ iff $\sigma?I \sqsubseteq i$. Note that if σ is an execution with input i and $i \sqsubseteq j$ then σ is also an execution with input j.

Executions in general may be only partially completed or they may be unrealistic because certain actions are systematically being ignored. For process networks, that cannot be tolerated and to be able to exclude such executions, we next define the notions of *maximality* and *fairness*.

DEFINITION 5.4. (MAXIMALITY) *Let* $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ *be an execution of the LTS. Execution* σ *with input* i *is called maximal iff it is infinite or in its last configuration only read actions on input channels from which all input of* i *has been consumed are possible, i.e., if* $|\sigma| = n$ *and* $s_n \xrightarrow{\alpha_n}$ *then* α_n *starts with* $c?a$ *for some* $c \in I$ *and* $\sigma?c = i(c)$.

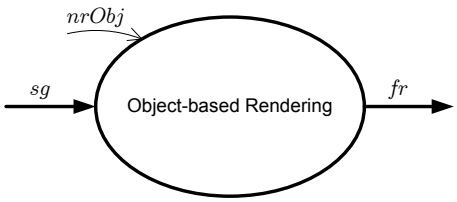

Figure 4: RPN Interface

Maximality states that the execution is complete, it has not been stopped at some arbitrary state while there is more input to be consumed or processed. Note that every finite execution ρ with input i of an LTS can be extended to a maximal execution with input i.

We can describe the externally observable behaviour of a labelled transition system by relating the output actions the LTS produces with the input provided to the network. In general, this gives a relation between input histories and output histories. One often restrict the attention to 'proper' executions in the sense that only maximal and fair executions are taken into account. (Executions are fair if no enabled actions are systematically being ignored and never take place.)

DEFINITION 5.5. (INPUT/OUTPUT RELATION) *The input output relation IO of an LTS is the relation* $\{(i, \rho!O) \mid \rho$ *is a maximal and fair execution with input* $i\}$.

As [4] argues, in general, the input/output relation is too abstract to adequately characterise the behaviour of an LTS. If it is non-deterministic, the order in which input is consumed or output is produced can be relevant. For Kahn process networks it is adequate, since the output only depends on the input and not on the order in which operations are executed. LTSs exhibiting such behaviour are called determinate and their input/output relation is a (continuous) function [6]. For our RPNs, the LTS cannot be determinate because of the introduction of events. This is why we resort to LTSs to define their semantics.

5.2 Reactive Process Networks

In this section, we present the details of what reactive process networks are. To describe the state of an RPN, we define a static part that remains the same during the lifetime of a network, and a dynamic part (configuration) that describes what is going on in the network at some point in time. Since RPNs have the ability to change their internal structure, the precise structure is part of the dynamic configuration. The static part of an RPN consists of its interface to the outside world, i.e. the set of (types of) ports through which it communicates. The input ports can be divided in streaming input ports and event input ports.

DEFINITION 5.6. (INTERFACE) *A* (process network) interface *is a triple* (SI, EI, O) *consisting of the disjoint sets* $SI \subseteq Chans$ *of* streaming input channels*,* $EI \subseteq Chans$ *of* event input channels *and* $O \subseteq Chans$ *of* output channels.

The interface of the rendering component in Figure 3 is $(\{sg\}, \{nrObj\}, \{fr\})$, graphically depicted in Figure 4.

To describe the dynamic configuration, we assume (to simplify presentation) that there is a universal, fixed set *Procs* of processes. With every process $p \in Procs$ is associated an interface (SI_p, EI_p, O_p) and a transition system

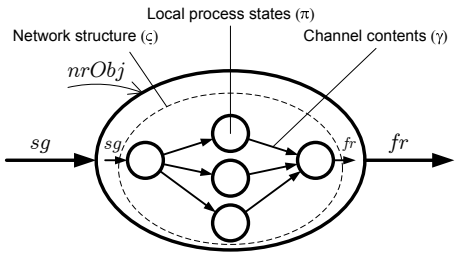

Figure 5: An RNP configuration

$\lambda_p = (S_p, \hat{s}_p, I_p, O_p, Act^*, \longrightarrow_p)$, where I_p is partitioned into the set SI_p of *stream* input channels and the set EI_p of *event* input channels. These processes can be Kahn processes, but they need not necessarily be.

DEFINITION 5.7. (CONFIGURATION) *A* (process network) configuration (ς, π, γ) *consists of*

- *a network structure* $\varsigma \in 2^{Procs} \times 2^{Chans}$ *that indicates which processes and channels are present and active in the current configuration; if* $\varsigma = (R, C)$*, then we use* $Procs(\varsigma)$ *and* $Chans(\varsigma)$ *to denote active processes* R *respectively the active channels* C*;*
- *a process configuration* $\pi : Procs(\varsigma) \to \bigcup_{p \in Procs(\varsigma)} S_p$ *is a function that assigns to every process* $p \in Procs(\varsigma)$*, a state* $\pi(p) \in S_p$*;*
- *a channel configuration* $\gamma \in H(Chans(\varsigma))$ *is a history function that assigns to every channel* $c \in Chans(\varsigma)$*, a finite string of tokens in* Σ_c*, representing the contents of the channels.*

Processes and channels are appropriately connected. This means that for every channel c *in* $Chans(\varsigma)$*, there is a process* p *in* $Procs(\varsigma)$ *that writes to it, i.e.* $c \in O_p$ *and a process* q *in* $Procs(\varsigma)$ *that reads from it, i.e.* $c \in I_q$*. The interface* (SI, EI, O) *of a structure consists of the set* SI *of stream input ports of processes in* ς *for which there is no channel in* $Chans(\varsigma)$ *connected to it, the set* EI *of event input ports not connected to a channel and the set* O *of output ports not connected to a channel. Two structures are called* interface compatible *if they have the same interface.*

The elements of a configuration are illustrated by Figure 5. We use *Confs* to denote the set of all configurations. Having interfaces and configurations, we can define what a reactive process network is.

DEFINITION 5.8. (RPN) *A* reactive process network (an RPN) *consists of*

- *an interface* (SI, EI, O)*;*
- *a set* $E \subseteq Confs \to Confs$ *of events; every event maps configurations to (interface compatible) configurations;*
- *an initial configuration* $(\hat{\varsigma}, \hat{\pi}, \hat{\gamma})$ *with interface* (SI, EI, O).

The interface is static. Events (may) cause a process network configuration, which includes its structure, to change. They roughly correspond to the transitions in the automata of the RPNs. These events E arrive on the event input

channels EI. Finally, the process network needs some initial configuration $(\hat{\varsigma}, \hat{\pi}, \hat{\gamma})$ to start from. Definition 5.8 is very liberal. It allows any change that doesn't affect the interface. How the effect of an event can be defined in practice is not elaborated in this paper. This may be small things like a new gain factor or new filter coefficients in some signal processing application, but it may also be a completely new function with a new network structure. In a programming system for RPNs, the latter type of event could correspond to a method or procedure that can create or destroy processes, channels or entire networks.

An example of such an RPN is the game application depicted in Figure 3 and discussed in Section 4. The streaming structure at the bottom of the figure (in a some state of execution) is a particular configuration. The automaton at the top is encoded in the event functions. Because they are functions, it is possible to deal with information that is still in the network (state information, or tokens in channels) in the transition to a new configuration so that this information is not lost. Hence, the configurations contain determine both the state of the automaton of the RPN and the current state of the processes of the current network. In the same way, the object-based rendering component can be reconfigured based on the number of objects that need to be rendered; an event $nrObj(2)$, for example, would reconfigure it to render 2 objects.

5.3 Operational Semantics

In this section, we define the operational semantics of a reactive process network, by associating with it a corresponding labelled transition system. An RPN consists of processes, which may in turn be other RPNs, or 'primitive' processes defined through other means. We construct a compositional semantics in the sense that when the labelled transition systems of the constituent processes are given, we construct a new labelled transition system for the whole RPN. This process is then repeated to inductively define the semantics of the an entire RPN.

Two types of things may happen to the network: data can be streaming through it, or it can encounter events that need to be processed. The combination of streaming and events will introduce non-determinism. The result however, should be as predictable as possible. In particular, we want to guarantee that input received before the arrival of a new event will lead to the required output, also if the output has not been completed when the event arrives. In implementations, this is achieved at the expense of a little extra synchronisation. The added predictability of the network's behaviour simplifies specifying the interaction between events and streams in our semantics. In specific subclasses of the RPN model, this synchronisation may be realised without much overhead.

A typical operational semantics of KPNs [5, 14] models streaming as a sequence of individual read and write actions of the processes involved in the computation of the output. Since, conceptually, the reaction of a process network to incoming data is immediate, it may not be disturbed by the processing of events. To this end, in our RPN semantics, these sequences of actions resulting from a data input are grouped together and represented as single, atomic transitions of the LTS. Effectively, this gives internal actions (i.e. completing the reaction to already received input) priority over processing of events. This leads us to the concept of

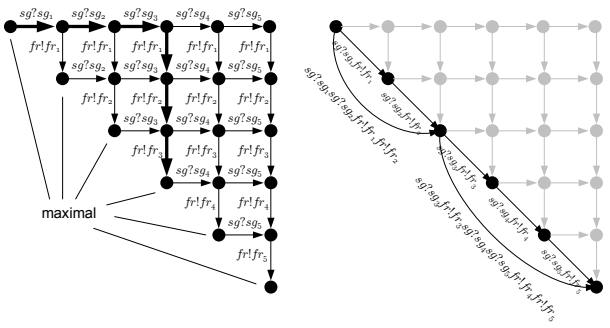

Figure 6: Streaming transactions

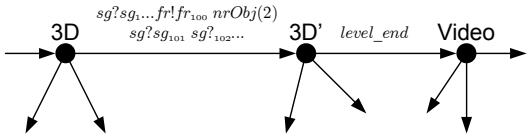

Figure 7: Execution of the 3D game RPN

so-called streaming transactions, as illustrated in Figure 6. The picture on the left shows the states and transitions of a process network performing individual input and output actions. Because of pipelining, the network can perform several input actions before the corresponding output actions are produced. (In an actual implementation the amount of pipelining may be restricted, but the semantics leaves all options open.) These actions are grouped together to form atomic transition, the streaming transactions, that end in states where events can safely be processed. For example, the sequence of transitions with thick arrows in the picture forms a streaming transaction; from the end state, only reading of new input is possible, all received input has been fully processed. (These states correspond to maximal executions and closely resemble the quiescent states of [16].) Some of the other streaming transactions are shown in the picture on the right.

The streaming transactions of an RPN are determined in two steps. Individual streaming transactions of the constituent processes are determined as well as the processing of events by these processes. Executions of these actions are then taken together to form maximal streaming transactions of the network. Such streaming transactions may not consume *input events*. However, they may include occurrences of internal events and hence they can be indeterminate. Figure 7 shows a part of a possible execution of the 3D game example. The first transition is a streaming transaction, consisting of streaming actions of the processes, but also internals events ($nrObj(2)$). Note that we have abstracted from internal actions of for instance the rendering component, which would also be visible in these transactions. The second transitions is an event, the player has reached the end of a level, and the game is reconfigured from 3D mode to the video mode. Such event transitions of an RPN are directly determined by the reception of input events, followed by the corresponding network transformation. The behaviour of the network as a whole is formed by interleaving transitions of both kinds as in the example. Events may lead to changes in the structure of a process network.

To make these concepts precise, the semantics of an RPN is described by (the maximal executions of) a labelled transition system:

$$(Confs, \hat{c}, SI \cup EI, O, Act^*, \longrightarrow)$$

The initial configuration $\hat{c} = (\hat{\varsigma}, \hat{\pi}, \hat{\gamma})$ is the initial configuration of the RPN.

As a first step to defining the transition relation $\longrightarrow$, we define the transition relation

$$\hookrightarrow \subseteq Confs \times Act^* \times Confs$$

which defines all internal streaming activity in terms of single transactions of its components. This transition relation is defined by the following induction rule in SOS (Structured Operational Semantics) notation.

$$\frac{\pi(p) \xrightarrow{\alpha}_p s,\ \alpha \in Act^*,\ \alpha?Chans(\varsigma) \sqsubseteq \gamma}{(\varsigma, \pi, \gamma) \overset{\alpha}{\hookrightarrow} (\varsigma, \pi\{s/p\}, (\gamma - \alpha?Chans(\varsigma)) \cdot \alpha!Chans(\varsigma))}$$

If process p can perform action(s) α from its current state $\pi(p)$, bringing it to state s, and the read actions of α match with the contents of the channels ($\alpha?Chans(\varsigma) \sqsubseteq \gamma$), then the network as a whole can perform action(s) α and in the configuration, local state of p ($\pi\{s/p\}$) and channel contents ($(\gamma - \alpha?Chans(\varsigma)) \cdot \alpha!Chans(\varsigma)$) are updated accordingly. Note that α can be a stream ($\alpha \in Act^*$, which may include actions internal to process p) or an event ($\alpha = c?e$, with $c \in EI_p$). This also means that internal action of the constituent processes are not abstracted and remain visible in the executions of the RPN. Such an abstraction would however be easy to add. In Figure 6, the internal actions of the rendering component are not shown.

A sequence of $\hookrightarrow$-transitions is called maximal if it is maximal with the input it has already consumed, i.e., $c_0 \overset{\alpha_0}{\hookrightarrow} c_1 \overset{\alpha_1}{\hookrightarrow} c_2 \overset{\alpha_2}{\hookrightarrow} \ldots c_n$ is maximal if for all α_n and c_{n+1} such that $c_n \overset{\alpha_n}{\hookrightarrow} c_{n+1}$, α_n starts with input actions from SI or EI.

Based on $\hookrightarrow$, we can now define the transition relation $\longrightarrow$ of the RPN by grouping together individual transactions of constituent processes into maximal streaming transactions of the RPN and defining the event transitions. First, the streaming actions.

$$\frac{c_0 \overset{\alpha_0}{\hookrightarrow} c_1 \overset{\alpha_1}{\hookrightarrow} c_2 \overset{\alpha_2}{\hookrightarrow} \ldots c_n \text{ is maximal}}{c_0 \xrightarrow{\alpha_0 \cdot \alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_{n-1}} c_n}$$

The $\hookrightarrow$ transitions correspond to the individual transitions in the left side picture of Figure 6. The atomic transitions to the transitions in the right side picture.

Next, we define the event transitions. External events result in a global transformation of the network structure and configuration.

$$\frac{e \in E,\ c \in EI}{(\varsigma, \pi, \gamma) \xrightarrow{c?e} e(\varsigma, \pi, \gamma)}$$

The semantics identifies events with functions that transform the configurations of the RPN. This leaves it open how these functions are specified or implemented in practice. It is unlikely that in an implementation, actual functions are being communicated. It is not hard to imagine, however, how an event can for instance be linked to a C/C++ function, associated with a specific event input port, that can make the necessary modifications to the process network, possibly dependent on parameters communicated with the event. [**Remark:** Refer to example with reconfiguration of rendering component.]

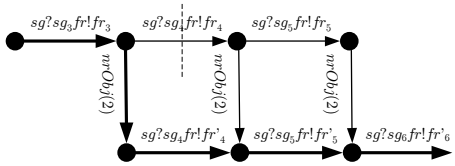

Figure 8: Prioritising actions

Together, the presented rules define the transition system that formally captures the behaviour of Reactive Process Networks. The intention is that this transition system defines the boundaries of how an RPN can be implemented. For specific implementations it may be desirable to impose further restrictions for more predictability, better performance or better timing behaviour.

5.4 Prioritising Events

The fact that the streaming transactions take arbitrary amounts of input can be interpreted as freedom to choose between processing events and processing new streaming input. In principle, this allows the execution to postpone processing of events for arbitrary periods of time. For the object-based renderer, if an event arrives indicating the requirement for extra processing elements for new objects, this event should be processed before any new scene graphs arrive which have more objects. If such a prioritisation of control messages is desired, it can be achieved by prioritising processing of events over the consumption of new streaming input. This is illustrated by Figure 8. The presented semantics allows for states that can both process an event ($nrObj(2)$) or continue processing of data (the streaming transitions labelled '$sg?sg_nfr!fr_n$'). Prioritisation of processing events can be taken care of by removing the streaming transitions for states where events are enabled. (In fact, disabling the transition has to be taken care of at the level of the process network in which this process is used, since the presence of the event transition merely denotes that the process *can* process the event *if there is an event available*. The latter can only be checked on the level of the process network.)

5.5 The Kahn Principle

A characteristic property of KPNs is known as the *Kahn Principle* [5, 14]. This principle states that the simple operational model of KPNs as processes reading tokens from and writing tokens to FIFO channels adheres to the denotational semantics of KPNs formulated as fixed-point equations on functions on strings of tokens communicated between processes. Essential to this property is that processes compute continuous functions on strings of tokens.

The extension of process networks with events makes that the individual processes no longer realise such continuous functions. It is well known [4] that a(n intuitive) denotational semantics for this case does not exist and hence, a principle similar to the Kahn Principle cannot be formulated. One can still prove though along the lines of [14] that as long as no events are being processed, the Kahn Principle

still holds. This informally translates to the idea that when there are no event activated, execution of the network can continue, unsynchronised as for KPNs, without disturbing the functional behaviour.

6. IMPLEMENTATION ISSUES

The operational semantics presented in this paper defined the boundaries of the behaviour that correct implementations of RPNs should adhere to. Within these boundaries there is still some room for making specific implementation decisions, depending on the application and the context. In this section, we briefly discuss some of these considerations as well as a prototype implementation we made.

6.1 Coordinating Streaming and Events

One of the most powerful aspects of KPNs is that execution can take place fully asynchronously. Processes need not synchronise and determinacy of the output is automatically guaranteed. The (accepted) drawback of our generalisation is that this advantage is (partially) lost. Before processing an input event, the streaming input to the network –conceptually– needs to be frozen and all data must be processed internally. Only when all data has been processed, the event can be applied and the data flow can be continued. If implemented in this way, the pipelining of the data flow may be disrupted and deadlines could potentially be missed because of this disruption if the nature of the application doesn't allow this. In practice, one can do better for many classes of systems. Instead of processing an event for the whole process network at once, it may in some cases be possible to make the changes along with the 'information flow'. In particular, if the response of a network to an event is the forwarding of the event to one or more of its subprocesses, then this forwarding can be synchronised with the flow of data such that pipelining need not be interrupted.

The presented operational semantics suggests that events must always be accepted by any process. In practice, it can be useful to allow a process some control w.r.t. the moment when events are accepted. For example, to allow it to accept events only at moments when the corresponding transformation is most easy to do, because the process is in a well-defined state, e.g. frame boundaries. Such an approach can for example be easily implemented if the underlying process network is an SDF graph and can hence be statically scheduled. It is then possible to define a cyclic schedule in such a way that one iteration of the cycle constitutes a single streaming transaction. Then, a test for newly arrived events can be inserted at the beginning of the cycle and event transitions can be safely executed.

6.2 Deadlock Detection and Resolution

The correct execution of KPNs using bounded FIFO implementations depends on the run-time environment to deal with artificial deadlock situations [17, 6]. Processes may be blocking because they are trying to write to a channel that does not have the space available to accept another token. If this situation turns into a cyclic dependency of processes, the capacity of one of the blocking channels needs to be increased. The same situation may arise in reactive process networks. We expect that we can deal with these artificial deadlocks in a similar manner as for ordinary KPNs [6]. Solving an artificial deadlock may be needed for completing the maximal transaction before processing an event.

```
class VideoFilter : public Process {
  public:
    VideoFilter(const Id& n, In<Frame>& frp,
        In<AlphaFrame>& ovlp, Out<Frame>& ofrp);

    /* streaming behaviour */
    void main();

    /* event handler */
    void newOverlay(AlphaFrame new_ovl);

  private:
    /* ports */
    InPort<Frame>            fr;
    EventInPort<AlphaFrame>  ovl;
    OutPort<Frame>           ofr;

    AlphaFrame current_ovl;

VideoFilter::VideoFilter(const Id& n,
        In<Frame>& frp, In<AlphaFrame>& ovlp, Out<Frame>& ofrp):
    Process(n),
    fr(id("fr"), frp),
    ovl(id("ovl"), ovlp, (void (Process::*)(AlphaFrame))
        &VideoFilter::setNewOverlay),
    ofr(id("ofr"), ofrp),
    current_ovl(new AlphaFrame)
{ }

void VideoFilter::main() {
    Frame processing_frame;

    while(true){
        fr.read(processing_frame);
        applyOverlay(processing_frame, current_ovl)
        out.write(processing_frame);
    }
}

void VideoFilter::newOverlay(AlphaFrame new_ovl) {
    current_ovl = new_ovl;
}
```

Figure 9: Fragment of Yapi specification with event processing

6.3 Experimental Implementation

In an experimental implementation, we have extended the YAPI [10] programming environment for implementing KPNs according to the presented semantics. Process networks, the hierarchical entities in YAPI, as well as the basic processes may receive events. We have chosen to use the priority model where further consumption of streaming input is stalled as soon as some event arrives to stimulate a quick response to events. All further internal actions are completed and after the process or network has been flushed in this way, the event is processed. After that, all input channels will be enabled again and streaming input can resume.

The functions that are associated with events in our semantics are implemented by member functions of the classes that implement the network or process receiving the event. This function is then linked to the event input port. Figure 9 shows, for instance, part of the definition of a video filter applying an overlay frame to all incoming video frames. The overlay frame can be adapted through events. The class definition defines the class `VideoFilter` as a process having a streaming input port `fr` for incoming frames, a streaming output port `ofr` for the output frames and an event input port `ovl`. The member function `main()` deals with streaming behaviour and the function `newOverlay()` is executed whenever a token arrives on the event input. The link between

the port and the event handler is defined with the construction of the port. The streaming behaviour of the filter is now a simple loop reading frames filtering the frame and writing the output frame. Independently, the event handler can update the current overlay frame. The run-time environment will automatically manage the synchronisation between streaming and even handling.

Because the behaviour of KPNs in YAPI cannot be analysed statically, a separate thread of execution is introduced to monitor all event inputs. The run-time environment then coordinates the execution of the streaming thread with the events thread. If we would introduce event handling in statically analysable applications, such as SDF graphs, then the event processing can be incorporated in the static schedule resulting in an efficient implementation.

7. CONCLUSIONS

In this paper, we have introduced Reactive Process Networks as a formal model of computation for stream-based applications with additional reactive behaviour. The model intends to cover the behaviour of streaming kernels as well as more dynamic, irregular streams and the enclosing control components. The formal model provides a sound basis for the construction of analysis tools as well as programming environments and APIs or synthesis tools for dynamic streaming systems. A semantics is given in terms of a labelled transition system that prescribes the possible orderings of read and write actions of the network. We have implemented an extension of YAPI, a programming environment for Kahn Process Networks originally developed at Philips Research, along the lines of the presented model.

Future work includes analysis and synthesis methods and tools that build upon this semantics. Attention should also be focussed on subsets of the model that allow for efficient implementation, such as Synchronous Data Flow models. Furthermore, implementation aspects such as the run-time deadlock detection and resolution mechanism have to be adapted to the RPN model. Important will also be timing analysis of RPNs. Response times for events need to be predictable, as well as throughput and latency of the streaming.

8. REFERENCES

[1] T. Basten and J. Hoogerbrugge. Efficient execution of process networks. In A. Chalmers, M. Mirmehdi, and H. Muller, editors, *Proc. of Communicating Process Architectures 2001, Bristol, UK, September 2001*, pages 1–14. IOS Press, 2001.

[2] B. Bhattacharya and S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, October 2001.

[3] S. Bhattacharyya, P. Murthy, and E. Lee. Synthesis of embedded software from synchronous dataflow specifications. *J. VLSI Signal Process. Syst.*, 21(2):151–166, 1999.

[4] J. Brock and W. Ackerman. Scenarios: A model of non-determinate computation. In J. Díaz and I. Ramos, editors, *Formalization of Programming Concepts, International Colloquium, Peniscola, Spain, April 19-25, 1981, LNCS Vol. 107*, pages 252–259. Springer Verlag, Berlin, 1981.

[5] A. Faustini. An operational semantics for pure dataflow. In M. Nielsen and E. M. Schmidt, editors, *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proc., LNCS Vol. 140*, pages 212–224. Springer Verlag, Berlin, 1982.

[6] M. Geilen and T. Basten. Requirements on the execution of Kahn process networks. In P. Degano, editor, *Proc. of the 12th European Symposium on Programming, ESOP 2003, Warsaw, Poland, April 7-11, 2003. LNCS Vol.2618*. Springer Verlag, Berlin, 2003.

[7] A. Girault, B. Lee, and E. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 18(6):742–760, June 1999.

[8] M. Goel. Process networks in Ptolemy II. Technical Memorandum UCB/ERL No. M98/69, University of California, EECS Dept., Berkeley, CA, December 1998.

[9] G. Kahn. The semantics of a simple language for parallel programming. In J. Rosenfeld, editor, *Information Processing 74: Proc. of the IFIP Congress 74, Stockholm, Sweden, August 1974*, pages 471–475. North-Holland, Amsterdam, Netherlands, 1974.

[10] E. Kock et al. YAPI: Application modeling for signal processing systems. In *Proc. of the 37th. Design Automation Conference, Los Angeles, CA, June 2000*, pages 402–405. IEEE, 2000.

[11] B. Lee. *Specification and Design of Reactive Systems.* PhD thesis, Electronics Research Laboratory, University of California, EECS Dept., Berkeley, CA, May 2000. Memorandum UCB/ERL M00/29.

[12] E. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL No. M01/11, University of California, EECS Dept., Berkeley, CA, March 2001.

[13] E. Lee and D. Messerschmitt. Synchronous data flow. *IEEE Proceedings*, 75(9):1235–1245, Sept. 1987.

[14] N. Lynch and E. Stark. A proof of the Kahn principle for Input/Output automata. *Information and Computation*, 82(1):81–92, 1989.

[15] A. Martin. The probe: An addition to communication primitives. *Information Processing Letters*, 20(3):125–130, 1985.

[16] S. Neuendorffer and E. A. Lee. Hierarchical reconfiguration of dataflow models. In *Proc. Second ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2004)*. IEEE Computer Society Press, 2004. to appear.

[17] T. Parks. *Bounded Scheduling of Process Networks.* PhD thesis, University of California, EECS Dept., Berkeley, CA, Dec. 1995.

[18] K. Strehl, et al. FunState - an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):524–544, Aug. 2001.

[19] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Grenoble, France, April 8-12, 2002, LNCS Vol. 2306*, pages 179–196. Springer Verlag, Berlin, 2002.