Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

Video Processing in occam-pi

Carl G. RITSON, Adam T. SAMPSON and Frederick R.M. BARNES

Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, England.
{cr49 , ats1 , frmb}@kent.ac.uk

Abstract. The occam-π language provides many novel features for concurrent software development. This paper describes a video processing framework that explores the use of these features for multimedia applications. Processes are used to encapsulate operations on video and audio streams; mobile data types are used to transfer data between them efficiently, and mobile channels allow the process network to be dynamically reconfigured at runtime. We present demonstration applications including an interactive video player. Preliminary benchmarks show that the framework has comparable overhead to multimedia systems programmed using traditional methods.

Keywords. occam-pi, concurrency, process networks, video, video processing

Introduction

This paper describes a video processing framework written in the occam-π language. Its design derives from the primary author's experiences developing for and using a number of open source video processing tools [1,2] and applications [3,4]. This work shows that not only is occam-π suitable for developing whole multimedia applications, but also offers advantages for the development of individual multimedia software components.

A video processing framework, or more generally a multimedia framework, is an API which facilitates the interaction and transfer of data between multimedia-handling software components. Almost all video processing systems are constructed within such frameworks, with distinct software components composed into pipelines or layers.

Software components with standardised interfaces can easily be modelled using object-oriented techniques, and as a consequence most existing frameworks are written in an object-oriented style. AviSynth [1] (C++), the DirectShow Filter Graph [5] (C++), Kamaelia [6] (Python) and GStreamer [7] (C/GLib [8]) are examples of this.

Communication between components is implemented either by direct calls to object (component) methods [1], or by interaction with buffers [5,7] (often called pins or pads) between components. The second approach can be directly parallelised, whereas the first requires the addition of buffers between parallel components. The method call model, without parallelism, is often preferred for interactive applications. Here a control component pushes or pulls data to or from the user, as it is easier to reason about the data currently being presented.

Neither of these approaches simplifies the design process, particularly in the presence of parallelism. For example, in order to create a circular ring of filters using method calls, one component must act as the initiator to avoid the system descending into infinite recursion. It is often difficult to reason about the correctness of systems produced using such methods.

The occam-π language, with semantics based on Hoare's CSP [9,10] and Milner's π-calculus [11] process algebras, offers a number of features which can be used to overcome these problems:

- The language's formal basis makes it possible to reason about the behaviour of concurrent processes and their interactions: for example, design patterns exist which guarantee freedom from deadlock [12].
- Mobile data types allow concurrent processes to safely and efficiently pass data around by reference, whilst avoiding race-hazard and aliasing errors.
- Mobile channel ends and dynamic process creation facilities allow networks of processes to be constructed, modified and destroyed at runtime.

The transputer processor [13], for which the occam language was originally developed, has previously been used for multimedia [14], and occam-like languages have been developed which allow the dynamic specification of multimedia systems [15]. However, such research predates the occam-π language and thus cannot take advantage of its new features [16,17] — occam-π's potential in this area is as yet unexplored.

The framework presented in this paper uses occam channels, grouped into occam-π channel types, to provide a standardised interface between components. Mobile data provides for efficient communication within the system, and mobile channel ends allow those systems to be reconfigured dynamically at run-time. The resulting process networks are similar to the models of traditional video processing systems, but have an implementation that more closely resembles the design.

Section 1 explores the development of an occam-π video player, looking at both 'push' and 'pull' modes of operation. Sections 2 and 3 examine the protocols and connectivity in detail. Issues of dynamic network reconfiguration are discussed in section 4, followed by an example application in section 5. Section 6 discusses the handling of status reporting in such dynamic networks. Initial conclusions and a discussion of future work are given in section 7.

1. ovp – The occam-π Video Player

This section explores the development of the occam-π video player, ovp, in order to give an insight into ideas underpinning the framework to be presented in the rest of the paper. Readers with a particular interest in the framework's implementation details may wish to skip ahead to section 2.

1.1. Basic Player

A basic video player needs to process and output two tracks of data (audio and video) concurrently. In order to achieve this we could use the process network shown in Figure 1. The specific decoder and output types can be inferred from the initial setup, `init`, messages which propagate through the network. The network post-`init` messages might look like Figure 2. This is in effect a form of runtime typing, and as such there is no guarantee that the given network of components will function together; section 7.2 discusses this further.

Data will flow through the network as the output processes consume it (at a rate dependent on timecodes embedded in the data flow). The network will act as a pipeline, with the decoders running in parallel with the output processes. This will continue until an `end` message is received, which will flush and shut down the network. This gives us linear playback.

1.2. User Control

A more general video player will have user control in the form of non-linear playback controls and pausing. For this we need to modify the network; an initial solution is presented in Figure 3. The "User Control" process repositions the input file in response to user seek requests. The purpose of the "Flow Control" processes is less obvious. As the network buffers data, a pause or change in track position will take time to filter through from the demulti-

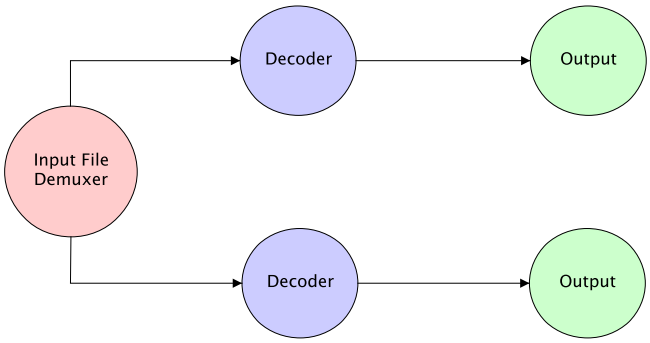

Figure 1. Process network for a simple occam-π video player.

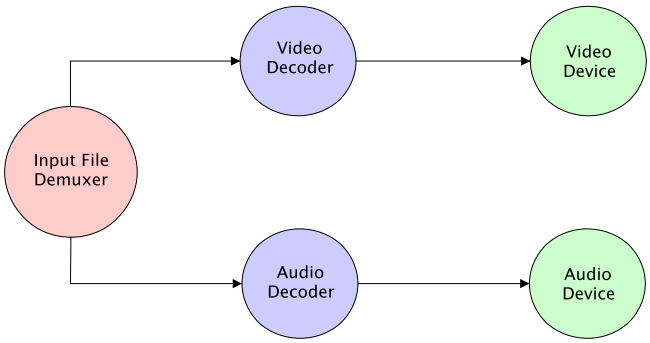

Figure 2. Simple occam-π video player after init messages have passed through the network.

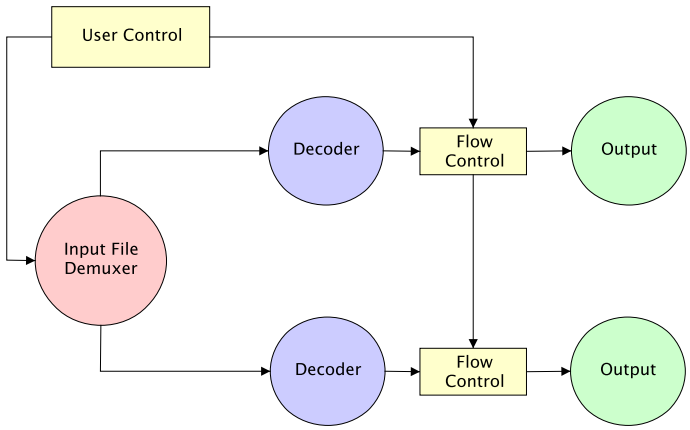

Figure 3. Process network for a seekable occam-π video player.

plexer to the outputs. This is not desirable if we want the network to appear responsive to user input. The flow controls are thus used to drain the input and decoding parts of the network during a pause or seek, and purge the outputs, meaning the user does not have to wait for the pipeline to empty before new data is presented.

One significant issue with this design is that it requires the temporal position of both streams to be the same after a seek or pause, otherwise there will be skew between them. This is not something we can guarantee. After a pause we will have introduced a skew proportional to the size of any output buffers (audio output devices always have buffers). For seeking there is no guarantee that the resolution of positioning data will be the same for all tracks. The timeline resolution of video will typically be several seconds (depending on the frequency of "key frames" upon which the decoder must synchronise), and audio hundreds of milliseconds (depending on packet size). Therefore, after a seek, the tracks will most likely be out of sync.

This problem can be resolved by making two changes:

1. When unpausing or seeking, decide a new position for all streams and distribute this to the flow controls, which then discard data before the new position.
2. Provide synchronisation facilities in the output processes.

1.3. Output Synchronisation

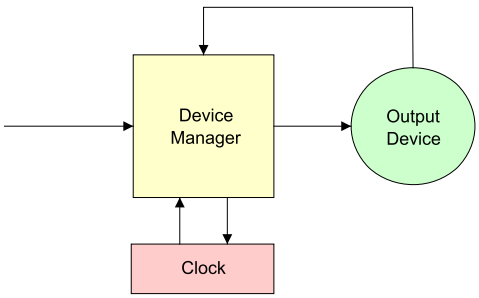

Figure 4. Breakdown of an output process, showing processes added to aid synchronisation.

In order to support output synchronisation, the output processes are broken down into three parts, as shown in Figure 4.

The embedded output device process acts in a pass-through manner. The device manager monitors the position of the output device using the timecodes of its output stream and delays frames appropriately using the clock process. The clock process converts the KRoC environment's microsecond timers to nanosecond timecodes. Given the KRoC environment's nanosecond communication times, reading the time via requests to separate processes should not lead to any significant inaccuracies, although it could be inefficient when used on a large scale.

The device manager starts in an unsynchronised state, and requests that the clock synchronise when it receives a timecoded message, providing the timecode as the synchronisation point. On receipt of a purge message, the device manager resets the clock and returns to an unsynchronised state. Synchronising all the outputs of the network is now done by synchronising their respective clocks (Figure 5).

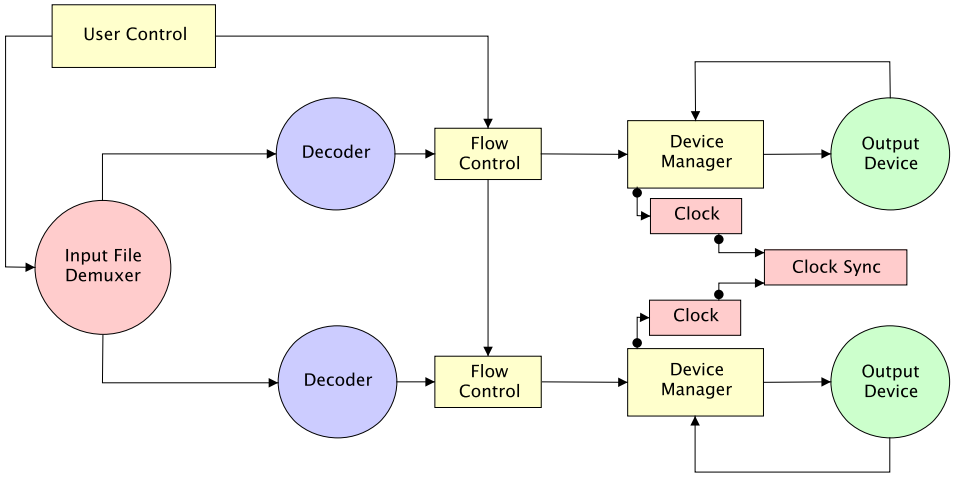

Figure 5. The occam-π video player, push-based seekable design with synchronised outputs.

Whenever a clock receives a synchronise or reset request, it forwards this to the "Clock Sync" process. The clock sync process in turn initiates a synchronisation cycle (if one is

not already running), which resets all other connected clocks. A clock which has been reset interrupts any pending alarms and refuses to accept new alarm requests until it has been resynchronised.

Once all clocks are attempting to synchronise and have presented the clock sync process with the desired timecode, the sync process picks the earliest timecode and associates it with a point in KRoC environment time. This association is the synchronisation point and is returned to all the clocks. All clocks thus acquire the same mapping of KRoC timer offset to timecode. This process is very much like a barrier. In practice the sync process returns a synchronisation point slightly earlier than that requested, allowing some propagation time.

A synchronisation mechanism similar to this could be extended to work across multiple distributed hosts, allowing the synchronisation of multiple distributed output devices – a topic for future research.

1.4. Pull Model

The design ideas so far presented only need employ the framework's stream protocol `P.MM` (see section 2). While these designs do work in practice, they are overly complex; as a side-effect of the input process driving the network, changes to the flow must be applied in two places. It makes more sense to have the process receiving user requests drive the network and hence be able to respond directly to user requests. For this we can use the framework's request/response protocols `P.MM.CTL` and `P.MM.SEEKABLE` (see section 3).

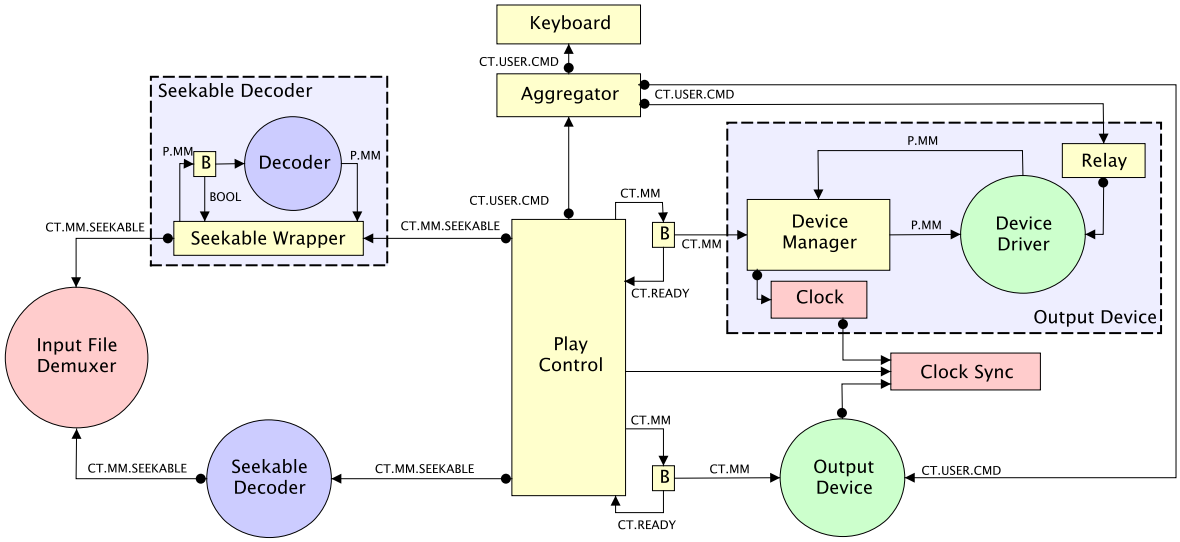

Figure 6. Process network for the complete occam-π video player, based on a pull model.

Figure 6 shows the operating process network for the complete occam-π video player, built using a request/response model. The "Play Control" process requests data from the inputs via the decoding pipelines, and passes it to the outputs, which are synchronised as previously described. The channel between the play control and the clock sync process is used to inform the clock sync process how many processes should be synchronised (some tracks may have come to an end and will not need synchronising). The seekable decoders are simply `decoder` processes extended to handle `P.MM.CTL/SEEKABLE` protocols using the seekable wrapper described in section 3.4. A flow path exists from each output back to the play control process via `CT.USER.CMD` channel bundles. This allows user commands input via the device, for example from an X11 window, to control playback.

An advantage of this design is that, since input tracks are considered separate streams and only share the common factor of the play control process, tracks need not come from

the same source. This means that audio from one file could be combined with video from another file without a complex backend synchronising the input processes. A “multi-play” mode in the occam-π video player uses this feature to play the video from any number of files simultaneously, synchronised in the same way as a pair of audio and video tracks.

Another advantage of the pull model is that by adding filters which intercept requests and distribute them over input sources, many separate input files could be arbitrarily combined into a single track (figure 7). This idea has not yet been implemented.

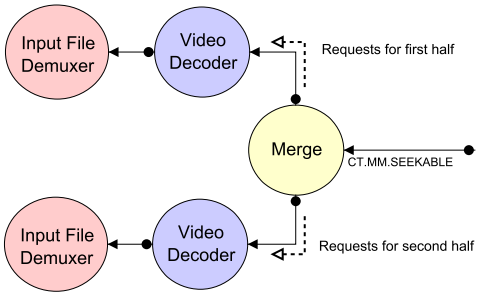

Figure 7. Possible design for a Merge process, which combines two other inputs.

2. Streaming – P.MM

At the heart of the framework is a single stream protocol `P.MM` (Protocol MultiMedia), which carries video and audio along with untyped “packet” data and commands using a “push” data flow model. The majority of the data elements are declared mobile [16]. If mobile data types were not used then data would need to be copied between communicating processes – highly inefficient for video, where the data-rate will typically exceed 20MB/sec (for standard-definition broadcast video). Using mobile data types, the pipeline of processes can be extended with no significant decrease in performance.

```
PROTOCOL P.MM
  CASE
    init;   MOBILE TRACK.INFO; MOBILE []BYTE
    packet; TID; MOBILE PKT.HDR; MOBILE []R.DESC; MOBILE []BYTE
    video;  TID; MOBILE VF.DESC; MOBILE []BYTE
    audio;  TID; MOBILE AF.DESC; MOBILE []BYTE
    flush
    purge
    skip
    end
:
```

init signals the start of a stream.

packet carries untyped media frames, typically compressed video or audio.

video carries a single video frame.

audio carries an audio frame of variable length.

flush instructs the receiver to output all ready buffers, then forward the flush command. This is necessary for the ideas presented in sections 3.4 and 1.

purge instructs the receiver to clear its internal state without generating any output, and to prepare for new data. The purge command is forwarded when the receiver is ready for new input. Like flush, this is used in sections 3.4 and 1.

skip instructs the receiver to do nothing. Unlike flush and purge it is not forwarded. This command is used to build zero-place buffers (see section 3.5).

end indicates the end of the stream; no data will follow. This provides a form of graceful termination [18]. On receipt the receiver should terminate after outputting any ready buffers (like a flush) and propagating the end message, unless it is restartable (such as `pluggable.stream.input.end` in section 5).

2.1. P.MM Usage Contract

There is as yet no language syntax for the specification of communication contracts in the occam-π language. For readability we have chosen to use a regular expression syntax for the contracts presented here. Hoare's CSP or a derivative such as that used in Singularity [19] could also have been used.

The expected sequence of messages on a channel of `P.MM` is as follows:

```
skip*
(init
  (packet|video|audio|flush|purge|skip)*
 )?
end
```

In summary, this means that the only certain event is the end of the stream, which can happen without prior initialisation. The `skip` command is permitted before initialisation to aid in the creation of zero-place buffers (see section 3.5).

Additionally, it is assumed there will not be a one-to-one mapping between input and output for processes implementing `P.MM` – a process may buffer as much or as little data as needed while maintaining FIFO ordering. The effect of this is that after sending an `init`, `purge` or `end` message to a process, it must not be assumed that the next output message will be of that type. Video encoders and decoders in particular require this form of internal buffering.

2.2. Timing – TID

Each elementary type (packet, video, audio) is associated with a `TID` data structure (Temporal IDentifier). A `TID` structure describes the position of a packet or frame within the timeline of the stream. This is done via the timecode field which is an offset in nanoseconds from a fixed point, typically the beginning of the stream. Nanoseconds are employed to allow the framework to manipulate data from Matroska [20] files without loss of timing resolution; however, microseconds would be sufficient for most present media formats. A duration in milliseconds is also stored, although this has limited uses.

```
DATA TYPE TID
  PACKED RECORD
    INT64 timecode:
    INT   duration:
:
```

Traditionally multimedia systems identify frames by their number in sequence from the beginning of the stream, or using SMPTE timecodes [21,22] which combine time and frame number offsets. This means that a stream is expected to have a fixed number of frames per unit time. In contrast, the framework presented here identifies frames purely based on time. There are three significant reasons for this:

1. When combining different streams together, it is more efficient to have a single common timeline to work with, rather than many sets of sample number and rate pairs

which must be normalised. Although at first this normalisation seems trivial, without timecode-based identification there is no way to synchronise and temporally manipulate streams without first knowing their respective sample rates – limiting the ways in which a system can dynamically adapt to new streams.
2. Any fixed sample rate system can be represented in a purely timecode-based system, assuming the timecodes have sufficient resolution and range.
3. A timecode-based system can represent streams with variable sample rates (discussed in the following subsection).

2.3. Aside on Variable-Frame-Rate Video

Although variable sample rate audio is uncommon, mixed frame rate video content is already in widespread circulation. In the production of NTSC television content and DVDs, it is common to use "pulldown" techniques to combine source material of different frame rates (typically 23.976fps for content that originated on film, and 29.970fps for content that originated on video). These mixed content streams can be represented more accurately in the digital domain by using a higher frame rate which is a common multiple of all the source rates (typically 119.880fps), and introducing "drop frames" where no actual frame data exists. This technique is, however, only applicable where there is a convenient common multiple between the frame rates.

An alternative is to convert the frame rates of the source materials; however, this conversion often introduces visible artefacts. Changing the frame rate of video can trivially be done by dropping or duplicating existing frames, but this causes jerky motion; to avoid this, it is necessary to synthesise new frames by estimating the motion of objects in the video images. Techniques to do this exist, but they are computationally very expensive and do not work well on "noisy" video.

A better option is to do away with the need for fixed frame rates, and just tag each frame with its corresponding time — this is *variable frame rate* (VFR) video. VFR avoids the need for resampling entirely, and allows the entire information content of the original video to be preserved.

VFR allows the time and rate of change to be free of quantisation. Modern compressed video formats are based around coding change — there is no need to code a new frame if nothing has changed. This is typically dealt with using drop frames, so a static image becomes a constant stream of "no change" messages. With VFR, there would be no output at all, offering potential bandwidth and disk space savings. Video scenes requiring smooth motion can have high rates of change, and other scenes lower rates. As the stream is not quantised, the actual changes can be placed at the most visually pleasing points in time, allowing acceptably smooth motion at lower data rates.

While existing output devices operate at a fixed frame rate, modern LCD displays are capable of running at rates far in excess of the captured rate of change in progressive video. This gives good scope for coding motion in a more visually pleasing way with less frames. It seems likely that future display devices will allow the display to be updated upon demand; they will be able to display VFR content without quantisation.

We feel that VFR has clear advantages over conventional constant-frame-rate video. Sample rates themselves result from the need to interface with analogue electronics, and as the world moves toward purely digital production and delivery of media content (high-definition television, TV over the Internet and digital end-to-end mastering), it is our expectation that variable frame rate material will become the standard. (It is worth noting that Internet streaming protocols are already beginning to support VFR content [23,24].)

2.4. Flexibility – TRACK.INFO

The TRACK.INFO data structure is heavily based on the "Track" descriptor in the Matroska [20] media container format. The Matroska format is designed to be able to hold an arbitrary number of media tracks of any type, and thus provided much of the inspiration for our framework's track-handling capabilities. Earlier versions of the TRACK.INFO were almost exact mirrors of the equivalent Matroska structure; however, the design has now been refined. The TRACK element of the name of this structure is itself a Matroska legacy; STREAM would be equally suitable.

3. Interactivity – P.MM.CTL/SEEKABLE

The following protocols act as a request/response pair and extend the commands in the stream P.MM protocol to provide interactive facilities through a "pull" model. This pull model sacrifices full parallel processing; only a single request is outstanding at a given time. It is intended for interactive applications where filling the pipeline with data is undesirable (due to the increase in end-to-end latency that results). Pre-roll buffer processes can be added to keep the pipeline full and restore parallel processing, if so desired.

3.1. Feedback – P.MM.CTL

P.MM.CTL is the request protocol. The process "pulling" data sends a single request and waits for a response.

```
PROTOCOL P.MM.CTL
  CASE
    init
    next
    seek; INT; INT64
    purge
    end
:
```

init requests the TRACK.INFO structure and setup data for the track.

next requests the logically next packet, video or audio frame in the stream.

seek requests that the stream be repositioned to a new timecode held in the INT64. The INT value is a constant describing how to handle cases where the exact timecode requested can not be reached, which is almost inevitable. If set to SEEK.CLOSEST then the closest match will be picked. SEEK.BACKWARD requests the closest point not after the specified timecode, and SEEK.FORWARD the closest point not before the specified timecode. In a typical video player, SEEK.BACKWARD will be used for rewind, and SEEK.FORWARD for fast-forward, in order to give the behaviour that a user would expect.

purge requests that the process clear all internal buffers. This request does not generate a response and is simply propagated to the preceding process.

end requests that the process terminate, and that it request the same of its preceding processes. Like purge, this does not generate a response.

3.2. Simplified P.MM – P.MM.SEEKABLE

P.MM.SEEKABLE is a simplified P.MM which carries responses. The data messages, packet, video and audio have the same meaning as in P.MM.

```
PROTOCOL P.MM.SEEKABLE
  CASE
    init;   MOBILE TRACK.INFO; MOBILE []BYTE
    packet; TID; MOBILE PKT.HDR; MOBILE []R.DESC; MOBILE []BYTE
    video;  TID; MOBILE VF.DESC; MOBILE []BYTE
    audio;  TID; MOBILE AF.DESC; MOBILE []BYTE
    seeked; INT64
    end
:
```

init no longer represents the start of stream; it simply describes the properties of the track and is the response to an init request.

seeked is sent in response to a seek request, and carries the timecode of the new stream position.

end means the end of the stream has been reached, but does not indicate that the component has terminated.

3.3. P.MM.CTL/SEEKABLE Usage Contract

The expected sequence of requests on a P.MM.CTL channel is as follows:

```
(init|next|seek|purge)* end
```

With the following request-response pattern:

```
init  => (init | end)
next  => (packet | video | audio | end)
seek  => (seeked | end)
purge => ()
end   => ()
```

3.4. Encapsulation

The P.MM and P.MM.CTL/SEEKABLE protocols were carefully designed to permit the construction of a wrapper around P.MM processes which presents a P.MM.CTL/SEEKABLE interface: seekable.wrapper (Figure 8).

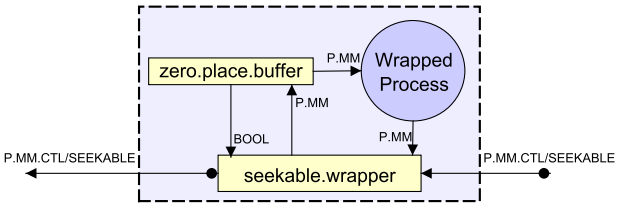

Figure 8. Process network for the seekable.wrapper.

1. At startup the wrapper requests an init message and other data from the preceding process, and feeds it to the wrapped process until an init message is produced. This init message is stored and used to respond to init requests from the successor process.

2. On a `next` request the wrapper makes `next` requests to the preceding process, feeding data to the wrapped process until output is produced. The output is forwarded to the successor. If an `end` response is encountered from the preceding process then the wrapped process is sent a `flush` message which causes it to output any available data. Should the wrapper receive a `flush` message from the wrapped process, it returns `end` to the successor.
3. On a `seek` request, the wrapped process is sent a `purge`. The wrapper waits for `purge` to be emitted by the wrapped process, discarding any intervening output. At the same time, the `seek` request is forwarded to the preceding process, and its response returned to the successor when the purge of the wrapped process is complete. This allows the successor to assume the process is ready for a new command immediately following a response. It should however be noted the performance impact of this design has not been explored.
4. A `purge` request is handled much like `seek` without any response to the successor.
5. An `end` request is forwarded to the preceding process and sent to the wrapped process. Once the wrapped process outputs `end`, then the wrapper terminates. This is the only point at which the wrapped process will receive an `end` message.

3.5. The zero.place.buffer

The seekable wrapper, along with many other processes in the framework, makes use of the `zero.place.buffer` process. It is a variant of the standard occam "requester" or "prompter" design pattern, using the `skip` message of the `P.MM protocol` to poll a process's ability to accept input. If a process accepts a `skip` message, it is "immediately" ready to accept any other command. Hence, the `zero.place.buffer` uses an explicit protocol command to interrogate a process rather than simply reporting when the process has accepted an already buffered message. A traditional "requester" will report when its (typically one-place) buffer is ready to be filled, whereas the `zero.place.buffer` reports when the process to which it is connected is ready. The `zero.place.buffer` is named for this lack of buffering.

4. Dynamism and Reconnectivity – CT.IO.CTL

While a multimedia system may be statically configured at compile time, it is often more useful to be able to dynamically reconnect its components as it is running. The following protocols provide a means to interrogate a component process about its connectivity, and to "plug" and "unplug" channel ends to create connections between processes at runtime. To facilitate this, a `P.MM` channel is placed in a `CT.MM` mobile record, and a pair of `P.MM.CTL` and `P.MM.SEEKABLE` channels are placed in a `CT.MM.SEEKABLE` channel type [17]. This method of building channel ends is a result of the design of the occam-π type system.

This dynamism allows the construction of very flexible file input processes. Until a file has been opened and its header parsed, it is not known what tracks it provides and their details; with these features, the tracks to be used can be selected at runtime, and an appropriate process network automatically constructed. While it would be possible to provide file input processes with fixed sets of channel parameters which are satisfied at runtime (these existed in earlier versions of the framework), it is more convenient to implement these on top of this generic architecture.

Processes implementing this interface are expected to operate in two modes, "started" and "stopped". A stopped process can have its connectivity interrogated and changed, whereas a started process can only be queried about its mode or stopped. It is expected that processes initially start in the stopped mode, as their connectivity is undefined.

4.1. P.IO.CTL.RQ

`P.IO.CTL.RQ` is the request component of the protocol.

```
PROTOCOL P.IO.CTL.RQ
  CASE
    inputs
    outputs
    start
    stop
    status
    end
    plug.mm.i;  INT64; CT.MM?
    plug.mm.o;  INT64; CT.MM!
    plug.mms.i; INT64; CT.MM.SEEKABLE!
    plug.mms.o; INT64; CT.MM.SEEKABLE?
    unplug.i;   INT64
    unplug.o;   INT64
:
```

inputs and outputs are used, respectively, to request details of the inputs and outputs of the process.

start and stop are used to change the mode of the process.

status queries the present mode of the process.

end closes down the `IO.CTL` interface; the process will terminate when *all* of its connected streams terminate.

plug.* and **unplug.*** messages are used to plug and unplug channel ends, with the `INT64` specifying the track number.

4.2. P.IO.CTL.RE

`P.IO.CTL.RE` is the response component.

```
PROTOCOL P.IO.CTL.RE
  CASE
    inputs;  MOBILE []INT; MOBILE []BOOL; MOBILE []TRACK.INFO
    outputs; MOBILE []INT; MOBILE []BOOL; MOBILE []TRACK.INFO
    plugged
    started
    stopped
    error;   INT
    unplugged.mm.i;  INT; CT.MM?
    unplugged.mm.o;  INT; CT.MM!
    unplugged.mms.i; INT; CT.MM.SEEKABLE!
    unplugged.mms.o; INT; CT.MM.SEEKABLE?
:
```

inputs and outputs return information describing the input and output connectivity of the process.

`[]INT` is an array of flags detailing the types of connections (stream or seekable), and whether a given connection must be plugged before the process can be started. For a process which filters data rather than originating it, the inputs and outputs will need to be plugged before the process can be started.

`[]BOOL` is an array of flags indicating whether a connection is plugged or unplugged. This is separate from the other flags as it changes during operation, where as the others typically do not.

`[]TRACK.INFO` is an array of the details of each connection. For inputs this will typically be a partial specification. An exact specification of what inputs the process will handle is not usually known in advance as most processes support various data formats. The framework currently supports only a simple form of partial specifications: unspecified values are given as zero.

plugged indicates a plug request was successful.

started and **stopped** indicate that the process has either started or stopped.

error indicates an error occurred during a start, stop, unplug or query of inputs or outputs. The `INT` value is a constant representing the reason for the error.

unplug.* messages are the response to a successful unplug request, or an unsuccessful plug request. The latter is required to prevent loss of the channel end which could not be plugged, in which case the `INT` value indicates why the plug failed.

It is anticipated that a process should check that inputs and outputs being plugged are compatible. This is not done in the framework at present as this verification, like the partial specification of track types, is beyond the scope of this research. Ideally the specifications of outputs should be updated as inputs are plugged; however, this would require sending and receiving from the inputs, which is forbidden when the process is stopped and would likely block as the other end may also be stopped. This might potentially be solved using a two phase system of input plugging followed by output plugging.

There is also a problem with the stopped and started modes of operation in that they introduce the possibility of deadlock. If we stop a process A to which another process B is attempting to communicate, then attempt to stop B, we are likely to deadlock as B may be blocked trying to communicate with A. This problem can be avoided by stopping processes in the same direction as the flow of data for `P.MM`, or the flow of requests for `P.MM.CTL`. It is, however, undesirable to have a system in which a sequence such as this must be adhered to, as it degrades the parallelism and is prone to implementation error.

One possible solution to the ordering and type checking problems would be to have a negotiation phase between the processes being connected and disconnected when connections are plugged and unplugged. This would allow both ends of a connection to know the state of the other end and thus not initiate communication while it was disconnected. Type negotiation would occur during the connection phase. This model would add a significant degree of complexity which we would rather avoid.

Further work is required to establish a simpler and safer model for reconnectivity than the one presented here, and to establish if wrappers like the seekable wrapper can be created for it.

5. dvplug – Digital Video Hot-plugging

The dvplug application (Figure 9) is a demonstration of digital video input, an encoding process, and the reconnectivity components of the framework. It responds both to user input and external events and demonstrates a possible real world application for the reconnectivity within the framework.

In the initial network, a clocked `test.card` source is wired through the control process (`record.ctl`) to the output. A `dv1394.plug.detector` process watches for new devices being connected to the host's IEEE1394 [25] bus, and reports their presence to the `player.spawner`.

On connection of a DV [26] device (such as a video camera), the `player.spawner` forks a set of processes to take input from the new device, decode it and remove the audio stream. Using the `CT.IO.CTL` protocol, it stops the `pluggable.stream.input.end`, disconnects

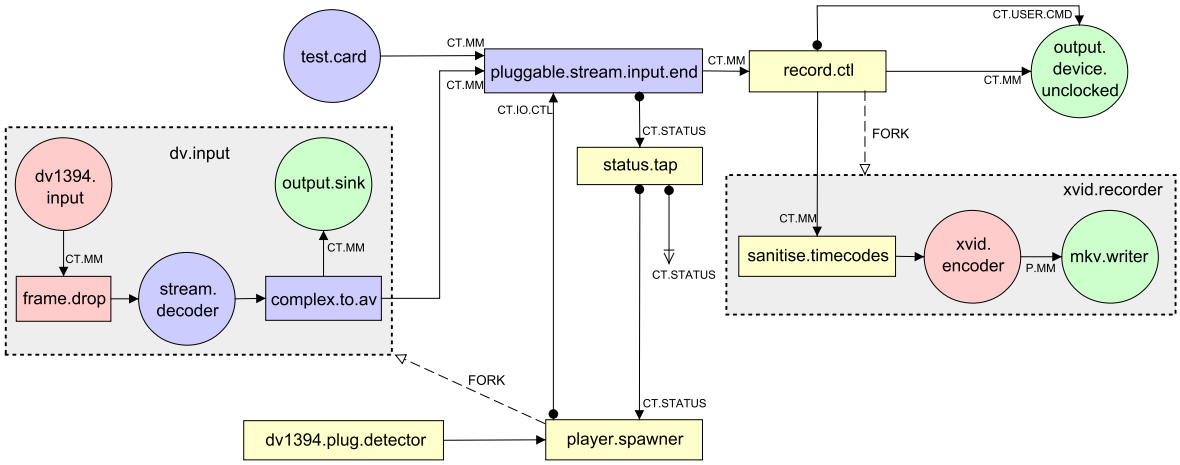

Figure 9. Process network for the dvplug application.

the `test.card` source from it and plugs in the new device's network. It then restarts the `pluggable.stream.input.end` and stores the `test.card` source channel end for later use.

While the DV input network is connected, the `player.spawner` ignores any messages about new devices. Using a `status.tap` (see section 6), it detects when the stream from the DV input network terminates (`pluggable.stream.input.end` does not propagate end messages). When such termination occurs the `player.spawner` disconnects the channel end for the terminated DV input network and reconnects the `test.card` source. Then after restarting the network it returns to monitoring events from the `dv1394.plug.detector`.

In parallel to this, the `record.ctl` process responds to record commands from the user, and transfers data from the `pluggable.stream.input.end` to the output. On receipt of a record command, it forks a recording network, and starts copying any received messages to it in addition to the regular output. When the user requests the end of recording, the recording network is sent an end message, and the channel end to it is discarded.

6. Status Reporting Backend

Every distinct component developed for the framework takes a `SHARED CT.STATUS!` channel end as a parameter. This provides a means for processes to output debugging information, errors and other events in a safe and uniform manner. The channel end is manipulated using a set of "`status.`" prefixed processes.

The status backend is implemented as a automatically growing N-way tree. Node creation requests are passed to the root which forks off the new `status.node` processes before passing connections to them back up the tree where they are "wired in". As messages pass down the tree to the root, they pick up the tags of all the nodes they pass through. By feeding the output of the root to a terminal it is easy to monitor the execution of the application network (particularly if `status.debug` calls are well placed). An example network instantiation is shown in Figure 10.

Another method for implementing a similar backend would be to use a flat structure with a root process which grows and shrinks an array of channel ends as nodes are created and destroyed. However such an approach would be inefficient as a result of resizing and `ALT`ing over the root array. It would also not be able to tag messages in a tree fashion as the presented implementation does, and would complicate process-to-process monitoring.

Other than user monitoring of the application, the status network can be used by processes within the network to monitor each other. This is done by inserting `status.tap` processes which copy, to an extra channel, a given subset of the messages passing through them.

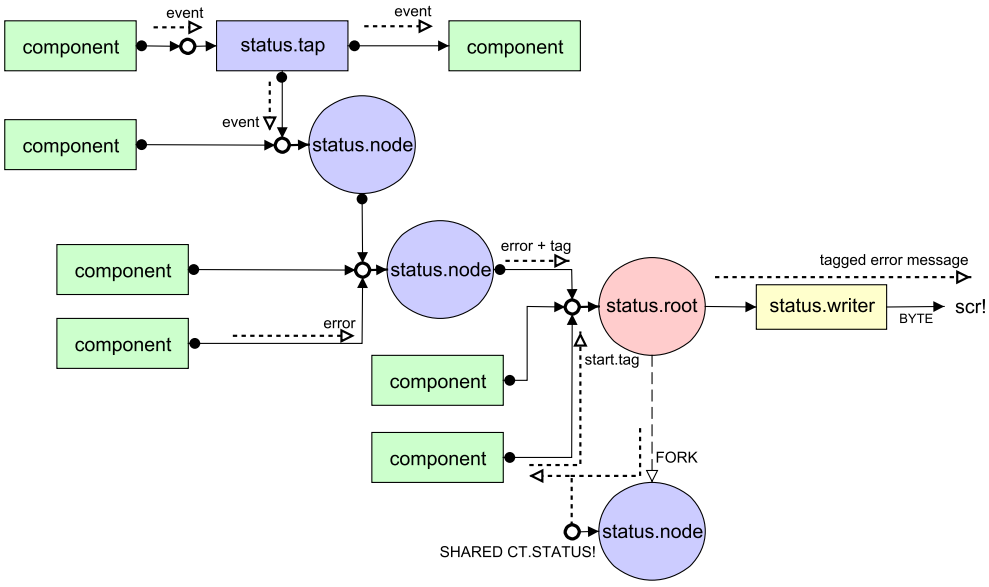

Figure 10. Example instantiation of the status backend.

This mechanism is used in the dvplug application (see section 5) to detect termination of DV input networks, using an `EOS` event.

At present the tags within the backend are the strings passed to `status.startup`; this could lead to name collisions in large networks. However as `status.node` forking is done at the root, there should be no problem allocating each node a unique number as it is forked. This could be used in place of or in addition to the textual name – an area for improvement in future versions of the framework.

7. Conclusions and Future Work

7.1. Benchmarks

Preliminary profiling of the occam-π Video Player (section 1) using a kernel-based profiler [27] suggests that the framework presents no significant overhead. Framework code, "ovp" in table 1, accounts for only 5% of execution time of the occam-π Video Player. The remainder is spent in external libraries, decoding ("libavcodec") and copying data ("libc").

Component	CPU samples	% CPU time
libavcodec.so.51.5.0	167526	77.4733
libc-2.3.5.so	35507	16.4204
ovp	**11264**	**5.2091**
libpthread-0.10.so	1461	0.6756
libX11.so.6.2	160	0.0740
ld-2.3.5.so	153	0.0708
libXv.so.1.0	139	0.0643
libXext.so.6.4	26	0.0120
libfaad.so.0.0.0	1	4.6e-04

Table 1. Profiling data for the occam-π Video Player (ovp).

Table 2 shows the profiling data obtained from MPlayer [3,28], a popular open-source video player. As the results show, the number of CPU cycles used by the framework is directly comparable to that used by MPlayer's core.

Component	CPU samples	% CPU time
libavcodec.so.51.5.0	156290	91.0818
mplayer	**11382**	**6.6331**
libc-2.3.5.so	2182	1.2716
ld-2.3.5.so	1044	0.6084
libpthread-0.10.so	379	0.2209
libX11.so.6.2	203	0.1183
libXv.so.1.0	60	0.0350
libXext.so.6.4	47	0.0274
libXcursor.so.1.0.2	3	0.0017
libdl-2.3.5.so	1	5.8e-04
libICE.so.6.3	1	5.8e-04
libfreetype.so.6.3.7	1	5.8e-04

Table 2. Profiling data for MPlayer.

The two differ in that MPlayer is able to decode compressed audio and video directly into its output buffers, whereas ovp must do a single copy operation at output time which accounts for 15% of its execution time ("libc" in table 1). This is a limitation of the current implementation, and could be solved by providing a mechanism for buffer recycling.

7.2. Future Work

As the value of any particular framework lies in the functionality it provides, one clear expansion of this work would be to develop more software components. The present framework is lacking any significant filtering components, so adding simple video operations (crop, scale, merge, split) would be a logical next step. Following this it would be interesting to explore writing spatial and temporal noise reduction filters, using the occam-π language, as these are common operations in broadcast systems and could take advantage of internal parallelism.

For video surveillance applications, it would be useful to provide processes for monitoring sets of video streams and reporting changes. Processes of this kind could also be used in the control of robots and other embedded devices.

Extending the input and output capabilities of the framework is also an interesting area; in particular, replacing the C components of the `avi.input` and `mkv.input` processes with occam-π. These components were largely written in C as the language has better facilities for handling complex data structures than the present occam-π. However, an attempt at an occam-π implementation could inform the future development of the language's data structure facilities.

One area left largely unsolved in the present framework design is safe reconnectivity. Reconnections need to be made both type-safe (in a high-level sense, ensuring data formats are the same) and deadlock-free. Although this safety is not guaranteed for static networks, the issue is mitigated by runtime checks. These are insufficient when a component's input can potentially change format mid-stream (as a result of a reconnection).

Deadlock is possible in the current reconnectivity model if processes are not stopped in the correct order (in the direction of data flow). This is not an issue in small networks where a single process is controlling the reconnection, but in a larger system many different processes may be modifying the network in parallel, in which case the ordering rules cannot easily be enforced.

In order to maintain the simplicity of the framework's process interfaces, we feel that reconnectivity will best be implemented using wrapper processes, concealing the inherent complexity from component developers.

It would also be desirable to provide a scripting language for the specification of networks of filters (as AviSynth [1] does) and a tool for editing such networks in real-time (like GraphEdit [29]), both of which might build on ideas presented in [30].

A final area for future development is error handling. The framework at present attempts to conceal and suppress errors: an error will typically cause termination of a filter or conversion of it to a null filter which does nothing. This is an unsatisfactory state of affairs for broadcast applications; the framework should instead be able to correct errors by allowing failed components to be replaced on-the-fly. One possibility is an exception-like model, where the failed process emits an exception message providing its present state (channel ends, track information, etc.). A control process would handle the exception, make any changes required, then invoke a replacement process using the provided state information.

7.3. Final Remarks

We have explored, for the first time, the use of the occam-π language's unique concurrency features for building a video processing framework and applications.

Multimedia systems must present a concurrent interface to the real world, and hence require some degree of internal concurrency. In systems developed using traditional methods, concurrency must often be simulated by explicit scheduling; in occam-π, concurrency can be expressed directly using language constructs.

occam-π's language features allow us to structure highly-concurrent programs such that they can be easily understood. Such programs can take full advantage of the concurrency features of modern processing hardware, and their safety properties can be reasoned about using formal methods.

We have demonstrated that our framework has comparable efficiency to multimedia systems developed in C using traditional, non-concurrent methods. This is the result of using occam-π's mobile data types, which allow safe, zero-copy data exchange between concurrent processes.

We have described a process-oriented approach to the construction of reconfigurable components using occam-π's mobile channel types. These components may be dynamically reconnected at runtime with minimal disruption to the rest of the system.

To conclude, we have shown that occam-π and process-oriented techniques offer several advantages to developers of multimedia systems when used in preference to traditional languages and design methods.

References

[1] AviSynth. Open source, scripted, video post-production tool. URL: `http://www.avisynth.org/`.
[2] VirtualDub. Open source video capture/processing utility for 32-bit Windows platforms. URL: `http://www.virtualdub.org/`.
[3] MPlayer. Open source multi-platform movie player. URL: `http://www.mplayerhq.hu/`.
[4] Video LAN Client. Open source cross-platform media player. URL: `http://www.videolan.org/`.
[5] Microsoft. DirectShow. URL: `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directshow/htm/directshow.asp`.
[6] BBC. Kamaelia. Open source concurrent Python based toolkit with multimedia components. URL: `http://kamaelia.sourceforge.net/Home`.
[7] GStreamer. Open source multi-media framework. URL: `http://www.gstreamer.net/`.
[8] GLib. C utility library and object system. URL: `http://www.gtk.org/`.
[9] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-13-153271-5.
[10] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0-13-674409-5.
[11] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN-10: 0521658691, ISBN-13: 9780521658690.

[12] P.H. Welch, G.R.R. Justo, and C.J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In R. Grebe, J. Hektor, S.C. Hilton, M.R. Jane, and P.H. Welch, editors, *Transputer Applications and Systems '93, Proceedings of the 1993 World Transputer Congress*, volume 2, pages 981–1004, Aachen, Germany, September 1993. IOS Press, Netherlands. ISBN 90-5199-140-1. See also: http://www.cs.kent.ac.uk/pubs/1993/279.

[13] Inmos Limited. *The Transputer Databook (2nd Edition)*. Inmos Limited, 1989. INMOS document number: 72 TRN 203 01.

[14] Tony King. Pandora: An Experiment in Distributed Multimedia. *Comp. Graph. Forum*, 11(3):23–34, 1992.

[15] David May and Henk L. Muller. Using Channels for Multimedia Communication. Technical report, University of Bristol, Department of Computer Science, February 1998.

[16] F.R.M. Barnes and P.H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 243–264, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[17] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part I. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 331–361, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[18] P.H. Welch. Graceful Termination – Graceful Resetting. In *Applying Transputer-Based Parallel Machines, Proceedings of OUG 10*, pages 310–317, Enschede, Netherlands, April 1989. Occam User Group, IOS Press, Netherlands. ISBN 90 5199 007 3.

[19] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J.R. Larus, and S. Levi. Language support for Fast and Reliable Message-based Communication in Singularity OS. In *Proceedings of EuroSys 2006*, Leuven, Belgium, April 2006. URL: http://www.cs.kuleuven.ac.be/conference/EuroSys2006/papers/p177-fahndrich.pdf.

[20] Matroska. Extensible open standard audio and video container format. URL: http://www.matroska.org/.

[21] American National Standards Institute. ANSI/SMPTE 12M-1986, "Television - Time and Control Code - Video and Audio Tape for 525-Line/60-Field Systems", January 1986.

[22] Society of Motion Picture and Television Engineers. SMPTE 12M-1999, "Television, Audio and Film - Time and Control Code", 1999.

[23] David Singer. Associating SMPTE time-codes with RTP streams, January 2006. A mechanism for associating SMPTE time-codes with media streams, in a way that is independent of the RTP payload format of the media stream itself. URL: http://www3.ietf.org/internet-drafts/draft-ietf-avt-smpte-rtp-01.txt.

[24] Colin Perkins and Stephan Wenger. RTP Timestamp Frequency for Variable Rate Audio Codecs, October 2004. IETF memo discussing the problems of audio codecs with variable external sampling rates. URL: http://www3.ietf.org/proceedings/05mar/IDs/draft-ietf-avt-variable-rate-audio-00.txt.

[25] IEEE. Std. 1394-1995 - IEEE standard for a high performance serial bus, August 1996. ISBN: 1-55937-583-3.

[26] Society of Motion Picture and Television Engineers. SMPTE 314M-1999, "Television - Data Structure for DV-Based Audio, Data and Compressed Video - 25 and 50Mb/s", 1999.

[27] OProfile. A system-wide profiler for Linux systems. URL: http://oprofile.sourceforge.net/.

[28] TUX Magazine. TUX 2005 Readers' Choice Award Winners. URL: http://www.tuxmagazine.com/node/1000151.

[29] Microsoft. GraphEdit. DirectX SDK tool for creating and debugging filter graphs. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directshow/htm/simulatinggraphbuildingwithgraphedit.asp.

[30] D.J. Beckett and P.H. Welch. A Strict occam Design Tool. In C.R. Jesshope and A. Shafarenko, editors, *Proceedings of UK Parallel '96*, pages 53–69. Springer-Verlag, July 1996. ISBN: 3-540-76068-7.

A. Diagram Notation