Communicating Process Architectures 2009
Peter Welch, Herman Roebbers and Tobe Announced (Eds.)
IOS Press, 2009

Mobile Escape Analysis for occam-pi

Frederick R.M. BARNES

School of Computing, University of Kent,
Canterbury, Kent, CT2 7NF. England.

F.R.M.Barnes@kent.ac.uk

Abstract. Escape analysis is the process of discovering boundaries of dynamically allocated objects in programming languages. For *object-oriented* languages such as C++ and Java, this analysis leads to an understanding of which program objects interact directly, as well as what objects hold references to other objects. Such information can be used to help verify the correctness of an implementation with respect to its design, or provide information to a run-time system about which objects can be allocated on the stack (because they do not 'escape' the method in which they are declared). For existing object-oriented languages, this analysis is typically made difficult by aliasing endemic to the language, and is further complicated by inheritance and polymorphism. In contrast, the occam-π programming language is a *process-oriented* language, with systems built from layered networks of communicating concurrent processes. The language has a strong relationship with the CSP process algebra, that can be used to reason formally about the correctness of occam-π programs.

This paper presents early work on a compositional escape analysis technique for *mobiles* in the occam-π programming language, in a style not dissimilar to existing CSP analyses. The primary aim is to discover the boundaries of mobiles within the communication graph, and to determine whether or not they *escape* any particular process or network of processes. The technique is demonstrated by analysing some typical occam-π processes and networks, giving a formal understanding of their mobile escape behaviour.

Keywords. occam-pi, escape analysis, concurrency, CSP

Introduction

The occam-π programming language [1] is a highly concurrent process-oriented language, derived from classical occam [2], in which systems are built from layered networks of communicating processes. The semantics of classical occam are based largely on those of Hoare's Communicating Sequential Processes (CSP) [3], an algebra that can be used to reason about the concurrent behaviour of occam programs [4,5].

To occam, occam-π adds new mechanisms and language constructs for data, channel and process mobility, inspired by Milner's π-calculus [6]. In addition occam-π offers a wealth of other features that allow the construction of dynamic and evolving software systems [7]. Some of these extensions, such as dynamic process creation, mobile barriers and channel-bundles, have already had CSP semantics defined for them [8,9,10], providing ways for formal reasoning about these. These semantics are sufficient for reasoning about most occam-π programs in terms of interactions between concurrent components, typically to guarantee the absence of deadlock, or refinement of a specification. However, these semantics do not adequately deal with *escape analysis* of the various mobile types, i.e. knowing in advance the *range of movement* of mobiles between processes and process networks.

The escape analysis information for an individual process or network of processes is useful in several ways:

- For checking design-level properties of a system, e.g. ensuring that *private* mobile data in one part of a system does not escape.
- For the implementation, as it describes the components *tightly coupled* by mobile communication — relevant in shared-memory systems, where pointers are communicated between processes, and for the breakdown of concurrent systems in distributed execution.

The remainder of this paper describes an additional *mobility* analysis for occam-π programs, in a style similar to the well-known *traces*, *failures* and *divergences* analyses of CSP [11]. Section 1 provides a brief overview of occam-π and its mobility mechanisms, in addition to current analysis techniques for occam-π programs. Section 2 describes the additions for mobile escape analysis, in particular, a new $mobility$ model. Section 3 describes how mobile escape analysis is performed for occam-π program code, followed by initial applications of this to occam-π systems in section 4. Related research is discussed in section 5, with conclusions and consideration for future work in section 6.

1. Occam-pi and Formal Analysis

The occam-π language provides a natural expression for concurrent program implementation, based on a communicating processes model as described by CSP. Whole systems are built from layered networks of communicating processes, which interact through a variety of synchronisation and communication mechanisms.

The primary mechanism for process interaction is through channel communication, where *two* processes synchronise (with the semantics of CSP *events*), and communicate data. The occam-π '`BARRIER`' type provides synchronisation between any number of processes, but allows no communication (although barriers can be used to provide safe access to shared data [12]). The barrier type is roughly equivalent to the general CSP event, though our implementation does not support *interleaving* — synchronisation between subsets of enrolled processes.

There are four distinct groups of mobile types in the occam-π language, that cover all of the occam-π mobility extensions. These are mobile *data*, mobile *channel-ends*, mobile *processes* and mobile *barriers*. The operational semantics of these vary depending on the type of mobile (described below).

Mobile variables, of all mobile types, are implemented primarily as *pointers* to *dynamically allocated* memory. To avoid the need for complex garbage collection (GC), strict aliasing rules are applied. For all mobile types, routines exist in the run-time system that allow these to be manipulated safely including: allocation, release, input, output, assignment and duplication.

1.1. Operational Semantics of Mobile Types

Mobile data exists largely for performance reasons. Ordinarily, data is communicated over occam-π channels using a *copying* semantics — i.e. the outputting process keeps its original data unchanged, and the inputting process receives a copy (overwriting a local variable or parameter). With large data (e.g. 100 KiB or more), the cost of this copy becomes significant, compared with the cost of the synchronisation. With *mobile* data, only a *reference* to the actual data is ever copied — a small fixed overhead [13]. However, in order to maintain the aliasing laws of occam (and to avoid parallel race-hazards on shared data), the outputting process must *lose* the data it is sending — i.e. it is *moved* to the receiving process. A '`CLONE`' operator exists for mobile data that creates a *copy*, for cases where the outputting process needs to retain the data after the output.

Mobile barriers allow synchronisation between arbitrary numbers of parallel processes. This has uses in a variety of applications, such as the simulation of complex systems [14], where barriers can be used to protect access to shared data (using a *phased* access pattern of global read then local write). When output by a process, a reference to a mobile barrier is *moved*, unless it is explicitly *cloned*, in which case the receiving process is *enrolled* on the barrier before the communication completes.

Mobile channel-ends refer to the end-points of *mobile channel bundles*. These are structured types that incorporate a number of ordinary channels. Unlike ordinary channels, however, these *mobile* channel-ends may be *moved* between processes — dynamically restructuring the process network. Mobile channel ends may be *shared* or *unshared*. Unshared ends are always *moved* on output. Shared channel-ends are always *cloned* on output. Communication on the individual channels inside a shared channel-end must be done within a '`CLAIM`' block, to ensure mutually exclusive access to those channels.

Mobile processes provide a mechanism for *process mobility* in occam-π [1]. Mobile processes are either *active*, meaning that they are connected to an environment and are running (or waiting for an event), or are *inactive*, meaning that they are disconnected from any environment and are free to be moved between processes. Like mobile data, there is no concept of a *shared* mobile process, though a mobile process may contain other mobiles (shared and unshared) as part of its internal state.

The rules for mobile assignment follow those for communication — in line with the existing laws of occam. For example, assuming '`x`' and '`y`' are integer ('`INT`') variables, the two following fragments of code are semantically equivalent:

```
x := y
```

≡

```
CHAN INT c:
PAR
  c ! y
  c ? x
```

This rule must be preserved when dealing with mobiles, whose references are either moved or duplicated, depending on the mobile type used. The semantics of communication are also used when passing mobile *parameters* to dynamically created (forked) processes [15] — *renaming* semantics are used for ordinary procedure calls.

1.2. Analysis of occam-pi Programs

Starting with an occam-π process, it is moderately straightforward to construct a CSP expression that captures the process's behaviour [4,5]. Figure 1 shows the traditional 'id' process and its implementation, that acts as a one-place buffer within a process network.

```
PROC id (CHAN INT in?, out!)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      out ! x
:
```

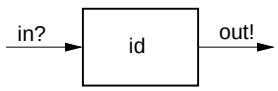


Figure 1. One place buffer process.

If the specification is for a *single* place buffer, this code represents the most basic implementation — all other implementations meeting the same specification are necessarily equivalent. The parameterised CSP equation for this process is simply:

$$\mathrm{ID}(in, out) = in \rightarrow out \rightarrow \mathrm{ID}(in, out)$$

This captures the behaviour of the process (interaction with its environment by synchronisation on 'in' and 'out' alternately), but makes no statements about individual data values. CSP itself provides only a limited support for describing the *stateful data* of a system. Where such reasoning is required, it would be preferable to use related algebras such as Circus [16] or CSP‖B [17].

Using existing and largely mechanical techniques, the traces, failures and divergences of this 'ID' process can be obtained:

$$\begin{aligned} traces\ \mathrm{ID} &= \{\langle\rangle, \langle in\rangle, \langle in, out\rangle, \langle in, out, in\rangle, \ldots\} \\ failures\ \mathrm{ID} &= \{(\langle\rangle, \{out\}), (\langle in\rangle, \{in\}), \\ &\qquad (\langle in, out\rangle, \{out\}), \\ &\qquad (\langle in, out, in\rangle, \{in\}), \ldots\} \\ divergences\ \mathrm{ID} &= \{\} \end{aligned}$$

As described in [11], the *traces* of a process are the sequences of events that it may perform. For the ID process, this is ultimately an *infinite* trace containing 'in' and 'out' alternatively.

The *failures* of a process describe under what conditions a process will *deadlock* (behave as $STOP$). These are pairs of traces and event-sets, e.g. (X, E), which state that if a process has performed the trace X and the events E are offered, then it will deadlock. For example, the first failure for the ID process states that if the process has not performed any externally visible events, and it is only offered 'out', then it will deadlock — because the process is actively only waiting for 'in'.

The *divergences* of a process are similar to failures, except these describe the conditions under which a process will *livelock* (behaves as div). The ID process is *divergence free*.

2. Mobility Analysis

The primary purpose of the extra analysis is to track the *escape* of mobile items from processes. With respect to mobile items, processes can:

- create new mobile items;
- transport existing mobiles through their interfaces; and
- destroy mobile items.

Unlike traces, failures and divergences, the $mobility$ of a process cannot be derived from a CSP expression of an occam-π process alone — requiring either the original code from which we would generate a CSP expression, or an augmented version of CSP that provides a more detailed representation of program behaviour, specifically the mobile operations listed above.

The remainder of this section describes the representation (syntax) used for mobility sequences, and some simple operations on these.

2.1. Representation

The mobility of a process is defined as a set of sequences of *tagged events*, where the events involved represent channels in the process's environment. For the non-mobile 'id' process discussed in section 1.2, this would simply be the empty set:

$$mobility\ \mathrm{ID} = \{\}$$

For a version of the 'id' process that transports mobile data items:

$$mobility\ \mathrm{MID} = \{\langle in?^{a}, out!^{a}\rangle\}$$

The name 'a' introduced in the mobility specification has scope across the *whole set* of sequences (though in this case there is only a single sequence) and indicates that the mobile data received from 'in' is the same as that output on 'out'. The direction (input or output) is relevant, since escape is asymmetric. Processes that create or destroy mobiles instead of transporting them are defined in similar ways.

The syntax for representing and manipulating mobility sequences borrows heavily from CSP [3,11], specifically the syntax associated with *traces*.

2.1.1. Shared Mobiles

For unshared mobile items, simple mobility sequences will have at most two items[1], reflecting the fact that a process acquires a mobile and then loses it — and therefore always in the order of an input followed by an output. For *shared* mobile items, mobility sequences may contain an arbitrary number of outputs, as a process can duplicate references to that mobile. Where there is more than one output, the order is unimportant — knowing that the mobile escapes is sufficient.

Shared mobiles are indicated explicitly — decorated with a '+'. For example, a version of the 'id' process that transports *shared* mobiles has the model:

$$mobility\ \mathrm{SMID} = \{\langle in?^{a+}, out!^{a+}\rangle\}$$

2.1.2. Client and Server Channel Ends

As described in section 1.1, mobile channel bundles are represented in code as pairs of connected ends, termed *client* and *server*. In practice these refer to the same mobile item, but for the purpose of analysis we distinguish the individual ends — e.g. for some mobile channel bundle '`a`', we use 'a' for the client-end and '$\bar{a}$' for the server-end. A version of 'id' that transports unshared server-ends of a particular channel-type would have the mobility model:

$$mobility\ \mathrm{USMID} = \{\langle in?^{\bar{a}}, out!^{\bar{a}}\rangle\}$$

These are slightly different from other mobiles in that they can appear as both superscripts (mobile items) and channel-names (carrying other mobile items). Recursive mobile channel-end structures can also carry themselves, expressed as, e.g. $\langle a!^{a}\rangle$.

Where there are multiple channels inside a mobile channel-end, the individual channels can be referred to by their index, e.g. $\langle a_{[0]}?^{x}\rangle$, $\langle a_{[1]}!^{a}\rangle$, to make clear which particular channel (for communication) is involved.

2.1.3. Undefinedness

In certain situations, that are strictly program errors, there is a potential for *undefined* mobile items to escape a process. Such an undefined mobile cannot be used in any meaningful way, but should be treated formally. A process that declares a mobile and immediately outputs it undefined, for example, would have the mobility model:

$$mobility\ \mathrm{BAD} = \{\langle out!^{\gamma}\rangle\}$$

The absence of such things can be used to prove that a process, or process network, does not generate any undefined mobiles.

[1]Higher order operations, e.g. communicating channels over channels, can produce mobility sequences containing more than two items — see section 3.7.

2.1.4. Alphabets

As is standard in CSP, we use sigma (Σ) to refer to the set of names on which a process can communicate. For mobility sequences, this can be divided into *output channels* ($\Sigma^!$) and *input channels* ($\Sigma^?$), such that $\Sigma = \Sigma^! \cup \Sigma^?$. Ordinary mobile items (data, barriers) are not part of this alphabet, mobile channel-ends are however.

The various channels that are in the alphabet of an occam-π process can also be grouped according to their type: Σ_t, where t is any valid occam-π protocol and $\mathbb{T}$ is the set of available protocols, such that $t \in \mathbb{T}$. Following on, $\Sigma_t = \Sigma^!_t \cup \Sigma^?_t$, and $\forall\, t : \mathbb{T} \cdot \Sigma_t \subseteq \Sigma$.

For referring to all channels that carry *shared* mobiles we have Σ_+, with $\Sigma_+ = \Sigma^!_+ \cup \Sigma^?_+$.

2.2. Operations on Mobility Sequences

For convenience, the following operations are defined for manipulating mobility sequences. To illustrate these, the name S refers to a set of mobility sequences, $S = \{R_1, R_2, \ldots\}$, each of which is a sequence of mobile actions, $R = \langle X_1, X_2, \ldots\rangle$. Each mobile action is either an input, $X_1 = C!^x$, or an output, $X_2 = D?^v$.

2.2.1. Concatenation

For joining mobility sequences:

$$\langle X_1, X_2, \ldots\rangle \hat{} \langle Y_1, Y_2, \ldots\rangle = \langle X_1, X_2, \ldots, Y_1, Y_2, \ldots\rangle$$

2.2.2. Channel Restriction

Used to remove named *channels* from mobility sequences:

$$\langle X_1, C!^x, \ldots\rangle - \{C\} = \langle X_1, \ldots\rangle$$

Note that this is not quite the same as *hiding*, the details of which are described later.

3. Analysing occam-pi for Mobility

This section describes the specifics of extracting mobile escape information for occam-π processes. Where appropriate, the semantics of these in terms of CSP operators are given. A *refinement* relation over mobility sets is also considered.

3.1. Primitive Processes

The two primitive CSP processes $STOP$ and $SKIP$ are expressed in occam-π using '`STOP`' and '`SKIP`' respectively. Although '`STOP`' is often not used explicitly, it is implicit in certain occam-π constructs — for example, in an '`IF`' structure, if none of the conditions evaluate to *true*, or in an '`ALT`' with no enabled guards. Both $SKIP$ and $STOP$ have empty mobility models. Divergence and chaos, for which there is no exact occam-π equivalent, have undefined though legal mobility behaviours — and are able to do anything that an occam-π process might.

$$mobility\ SKIP = \langle\rangle$$
$$mobility\ STOP = \langle\rangle$$
$$mobility\ \mathsf{div} = mobility\ CHAOS =$$
$$\{\langle C!^a\rangle \mid C \in \Sigma^!\} \cup \{\langle D?^x\rangle \mid D \in \Sigma^?\} \cup$$
$$\{\langle C?^v, D!^v\rangle \mid \forall\, t : \mathbb{T} \cdot (C, D) \in \Sigma^?_t \times \Sigma^!_t)\}$$

The models of divergence and chaos specify that the process may output *defined* mobiles on any of its output channels, consume mobiles from any of its input channels, and forward mobiles from any of its input channels to any of its output channels (where the types are compatible). However, neither divergence or chaos will generate (and output) *undefined* mobiles, but may forward undefined mobiles if these were ever received.

3.2. Input, Output and Assignment

Input and output are the basic building blocks of mobile escape in occam-π — they provide the means by which mobile items are *moved*. For example, a process that generates and outputs a mobile (which escapes):

```
PROC P (CHAN MOBILE THING out!)
  MOBILE THING x:
  SEQ
    ...  initialise 'x'
    out ! x
:
```

$$mobility\ \mathrm{P} = \{\langle out!^x\rangle\}$$

Correspondingly, a process that consumes a mobile:

```
PROC Q (CHAN MOBILE THING in?)
  MOBILE THING y:
  SEQ
    in ? y
    ...  use y
:
```

$$mobility\ \mathrm{Q} = \{\langle in?^y\rangle\}$$

A similar logic applies to assignment, based on the earlier equivalence with communication. For example:

```
PROC R (CHAN MOBILE THING in?, out!)
  MOBILE THING v, w:
  SEQ
    in ? v
    w := v
    out ! w
:
```

$$mobility\ \mathrm{R} = \{\langle in?^v, Lc!^v\rangle, \langle Lc?^w, out!^w\rangle\} \setminus \{Lc\}$$

The local channel-name Lc comes from the earlier model for assignment (as a communication between two parallel processes). The semantics for parallelism and hiding are described in the following sections. A compiler does not need to model assignment directly in this manner, however — it can track the movement of mobiles between local variables itself, and generate simpler (but equivalent) mobility sequences. For the above process 'R':

$$mobility\ \mathrm{R} = \{\langle in?^u, out!^u\rangle\}$$

3.3. Sequential Composition

Sequential composition provides one mechanism by which a mobile received on one channel can escape on another. In the case of the 'id' process, whose mobility model is intuitively obvious (but best determined automatically by a compiler or other tool):

```
SEQ
  in ? v
  out ! v
```

$$mobility\ \mathrm{ID} = \{\langle in?^{v}, out!^{v}\rangle\}$$

In general, the mobility model for sequential processes, i.e. $mobility(P\ ;\ Q)$, is formed by combining input sequences from $mobility\ P$ with output sequences from $mobility\ Q$, matched by the particular mobile variable input or output. When combining processes in this and other ways, the individual *variables* representing mobile items may need to be renamed to avoid unintentional capture.

3.4. Choice

Programs may make choices either internally (e.g. with 'IF' and 'CASE') or externally (with an 'ALT' or 'PRI ALT'). The rules for internal and external choice are straightforward — simply the union of the sets representing the individual choice branches. For example:

```
PROC plex.data (CHAN MOBILE THING in0?, in1?, out!)
  WHILE TRUE
    MOBILE THING v:
    ALT
      in0 ? v
        out ! v
      in1 ? v
        out ! v
:
```

$$mobility\ \mathrm{PD} = \{\langle in0?^{a}, out!^{a}\rangle, \langle in1?^{b}, out!^{b}\rangle\}$$

In general:

$$mobility\ (P \,\Box\, Q) = (mobility\ P) \cup (mobility\ Q)$$
$$mobility\ (P \sqcap Q) = (mobility\ P) \cup (mobility\ Q)$$

3.5. Interleaving and Parallelism

Interleaving and parallelism, both specified by 'PAR' in occam-π, have straightforward mobility models. For example, a 'delta' process for SHARED mobile channel-ends, that performs its outputs in parallel:

```
PROC chan.delta (CHAN SHARED CT.FOO! in?, out0!, out1!)
  WHILE TRUE
    SHARED CT.FOO! x:
    SEQ
      in ? x
      PAR
        out0 ! CLONE x
        out1 ! CLONE x
:
```

$$mobility\ \mathrm{CD} = \{\langle in?^{a+}, out0!^{a+}\rangle, \langle in?^{b+}, out1!^{b+}\rangle\}$$

This captures the fact that a mobile input on the 'in' channel escapes to both the output channels, indistinguishable from a non-interleaving process that makes an internal *choice* about where to send the mobile. In general:

$$mobility\ (P \parallel Q) = (mobility P) \cup (mobility\ Q)$$

Interleaving (e.g. $P \mid\mid\mid Q$) is a special form of the more general alphabetised parallelism, therefore it is not of huge concern for mobile escape analysis.

3.6. Hiding

Hiding is used to model the declaration and scope of channels in occam-π. In particular, it is also responsible for collapsing mobility structures — by removing channel names from them. Where occam-π programs are concerned, channel declarations typically accompany '`PAR`' structures. For example:

```
PROC network (CHAN MOBILE THING in?, out!)
  CHAN INT c:
  PAR
    thing.id (in?, c!)
    thing.id (c?, out!)
:
```

$$mobility\ \mathrm{NET} = \{\langle in?^a, c!^a\rangle, \langle c?^b, out!^b\rangle\} \setminus \{c\}$$

This reduces to the set:

$$mobility\ \mathrm{NET} = \{\langle in?^a, out!^a\rangle\}$$

The general rule for which is:

$$\begin{aligned} mobility\ (P \setminus x) = \big\{ & M \frown N[\alpha/\beta] \mid \\ & \big(M \frown \langle x!^\alpha\rangle, \langle x?^\beta\rangle \frown N\big) \in mobility\ P \times mobility P\big\} \cup \\ & \big((mobility\ P) - \big(\big\{F \frown \langle x!^\alpha\rangle \mid F \frown \langle x!^\alpha\rangle \in mobility\ P\big\} \\ & \quad \cup \big\{\langle x?^\beta\rangle \frown G \mid \langle x?^\beta\rangle \frown G \in mobility\ P\big\}\big)\big) \cup \\ & \big\{H \mid (H \frown \langle x!^\alpha\rangle) \in mobility\ P \wedge (\langle x?^\beta\rangle \frown I) \notin mobility\ P \wedge H \neq \langle\rangle\big\} \cup \\ & \big\{J \mid (\langle x?^\beta\rangle \frown J) \in mobility\ P \wedge (J \frown \langle x!^\alpha\rangle) \notin mobility\ P \wedge J \neq \langle\rangle\big\} \end{aligned}$$

The above specifies the joining of sequences that end with outputs on the channel x with sequences that begin with inputs on the channel x. The matching sequences are removed from the resulting set, however, the starts of unmatched output sequences and the ends of unmatched input sequences are preserved.

3.7. Higher Order Communication

So far, only the transport of mobiles over *static* process networks has been considered. However, in many real applications, mobile channels will be used to setup connections between processes, which are later used to transport other mobiles (including other mobile channel-ends). Assuming that the '`CT.FOO`' channel-type contains a single channel named '`c`', itself carrying mobiles, we might write:

```
PROC high.order.cli (CHAN CT.FOO! in?)
  CT.FOO! cli:
  MOBILE THING v:
  SEQ
    in ? cli
    ...  initialise 'v'
    cli[c] ! v
:
```

$$mobility\ \mathrm{HOC} = \left\{\langle in?^{a}, a!^{b}\rangle\right\}$$

This captures the fact that the process emits mobiles on the bound name 'a', which it received from its '`in`' channel. The type '`CT.FOO!`' specifies the client-end of the mobile channel[2]. A similar process for the server-end of the mobile channel could be:

```
PROC high.order.svr (CHAN CT.FOO? in?)
  CT.FOO? svr:
  MOBILE THING x:
  SEQ
    in ? svr
    svr[c] ? x
    ...  use 'x'
:
```

$$mobility\ \mathrm{HOS} = \left\{\langle in?^{\bar{c}}, \bar{c}?^{d}\rangle\right\}$$

Connecting these in parallel with a generator process (that generates a pair of connected channel-ends and outputs them), and renaming for parameter passing:

```
PROC foo.generator (CHAN CT.FOO! c.out!, CHAN CT.FOO? s.out!)
  CT.FOO? svr:
  CT.FOO! cli:
  SEQ
    cli, svr := MOBILE CT.FOO
    PAR
      c.out ! cli
      s.out ! svr
:

CHAN CT.FOO! c:
CHAN CT.FOO? s:
PAR
  foo.generator (c!, s!)
  high.order.cli (c?)
  high.order.svr (s?)
```

$$mobility\ \mathrm{FG} = \left\{\langle c.out!^{x}\rangle, \langle s.out!^{\bar{x}}\rangle\right\}$$

$$\begin{aligned} mobility &= \left\{\langle c!^{x}\rangle, \langle s!^{\bar{x}}\rangle, \langle c?^{a}, a!^{b}\rangle,\right. \\ &\qquad \left.\langle s?^{\bar{c}}, \bar{c}?^{d}\rangle\right\} \setminus \{c, s\} \\ &= \left\{\langle x!^{b}\rangle, \langle \bar{x}?^{d}\rangle\right\} \end{aligned}$$

This indicates a system in which a mobile is transferred internally, but never escapes. As such, we can hide the mobile channel event 'x' (also '$\bar{x}$'), giving an empty mobility set — concluding that no mobiles escape this small system, as we would have expected.

3.8. Mobility Refinement

The previous sections have illustrated a range of mobility sets for various processes and their compositions. Within CSP and related algebras is the concept of *refinement*, that operates on the traces, failures and divergences of processes, and can in general be used to test whether a particular implementation meets a given specification. In general, we write $P \sqsubseteq Q$ to mean that P *is refined by* Q, or that Q is *more deterministic* than P.

[2]The variable '`cli`' is a mobile channel bundle containing just one channel (named '`c`'), identified by a record subscript syntax: `cli[c]`.

For mobile escape analysis, it is reasonable to suggest that there may be a related *mobility refinement*, whose definition is:

$$P \sqsubseteq_M Q \qquad \equiv \qquad mobility\ Q \subseteq mobility\ P$$

The interpretation of this is that Q "contributes less to mobile escape" than P, and where the subset relation takes account of renaming within sets. This is not examined in detail here (an item for future work), but on initial inspection appears sensible — e.g. to test whether a specific implementation meets a general specification.

4. Application

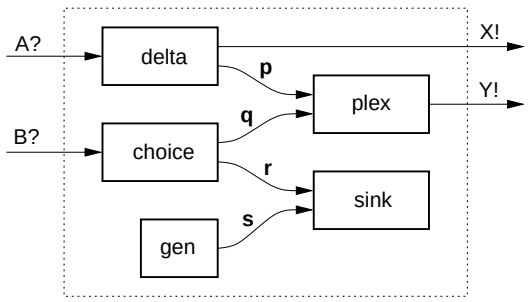
As previously discussed, the aim of this analysis is to determine what mobiles (if any) escape a particular network of occam-π processes, and if so, how they escape with respect to that process network (i.e. on which input and output channels).

Two examples of the technique are discussed here, one for static process networks and one for dynamically evolving process networks. The former is more typical of small-scale systems, such as those used in small (and memory limited) devices.

4.1. Static Process Networks

Figure 2 shows a network of parallel processes and the code that implements it. The individual components have the following mobile escape models:

$$mobility\ \mathrm{delta} = \{\langle in?^a, out0!^a\rangle, \langle in?^b, out1!^b\rangle\}$$
$$mobility\ \mathrm{choice} = \{\langle in?^a, out0!^a\rangle, \langle in?^b, out1!^b\rangle\}$$
$$mobility\ \mathrm{gen} = \{\langle out!^a\rangle\}$$
$$mobility\ \mathrm{plex} = \{\langle in0?^a, out!^a\rangle, \langle in1?^b, out!^b\rangle\}$$
$$mobility\ \mathrm{sink} = \{\langle in0?^a\rangle, \langle in1?^b\rangle\}$$




```
PROC net (CHAN MOBILE THING A?, B?,
                            X!, Y!)
  CHAN MOBILE THING p, q, r, s:
  PAR
    delta (A?, X!, p!)
    choice (B?, q!, r!)
    gen (s!)
    plex (p?, q?, Y!)
    sink (r?, s?)
:
```

Figure 2. Parallel process network.

When combined, with appropriate renaming for parameter passing (and to avoid unintentional capture), this gives the mobility set:

$$mobility\ \mathrm{Net} = \{\langle A?^a, X!^a\rangle, \langle A?^b, p!^b\rangle, \langle B?^c, q!^c\rangle, \langle B?^d, r!^d\rangle,$$
$$\langle s!^e\rangle, \langle p?^f, Y!^f\rangle, \langle q?^g, Y!^g\rangle, \langle r?^h\rangle, \langle s?^h\rangle\} \setminus \{p, q, r, s\}$$

Applying the rule for hiding to the channels p, q, r and s gives:

$$\xrightarrow{\setminus\{p\}} \big\{\langle A?^a, X!^a\rangle, \langle A?^b, Y!^b\rangle, \langle B?^c, q!^c\rangle, \langle B?^d, r!^d\rangle, \langle s!^e\rangle, \langle q?^g, Y!^g\rangle, \langle r?^h\rangle, \langle s?^h\rangle\big\}$$

$$\xrightarrow{\setminus\{q\}} \big\{\langle A?^a, X!^a\rangle, \langle A?^b, Y!^b\rangle, \langle B?^c, Y!^c\rangle, \langle B?^d, r!^d\rangle, \langle s!^e\rangle, \langle r?^h\rangle, \langle s?^h\rangle\big\}$$

$$\xrightarrow{\setminus\{r\}} \big\{\langle A?^a, X!^a\rangle, \langle A?^b, Y!^b\rangle, \langle B?^c, Y!^c\rangle, \langle B?^d\rangle, \langle s!^e\rangle, \langle s?^h\rangle\big\}$$

$$\xrightarrow{\setminus\{s\}} \big\{\langle A?^a, X!^a\rangle, \langle A?^b, Y!^b\rangle, \langle B?^c, Y!^c\rangle, \langle B?^d\rangle\big\}$$

The resulting mobility analysis indicates that mobiles input on A escape through output on X and Y, and that inputs received on B either escape through Y or are consumed internally. The fact that certain mobility sequences are not present in the result provides more information: that mobiles input on A are never discarded internally, and that the resulting network does not generate escaping mobiles.

4.2. Dynamic Process Networks

In dynamically evolving systems, RMoX in particular [18,19], connections are often established within a system for the sole purpose of establishing future connections. An example of this is an application process that connects to the VGA framebuffer (display) device via a series of other processes, then uses that new connection to exchange mobile data with the underlying device. Figure 3 shows a *snapshot* of connected graphics processes within a running RMoX system.

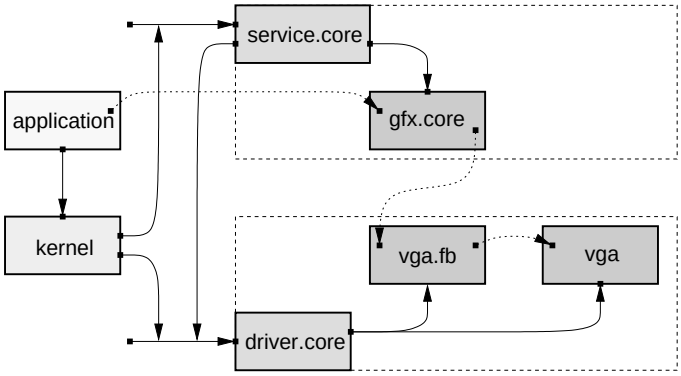


Figure 3. RMoX driver connectivity.

Escape analysis allows for certain optimisations in process networks such as these. If the compiler (and associated tools) can determine that mobile data generated in '`vga`' or '`vga.fb`' is not discarded internally, nor escapes through the processes '`gfx.core`' and '`application`', then it will be safe to pass the real framebuffer (video) memory around for rendering. Without the guarantees provided by this analysis, there is a danger that parts of the video memory could escape into the general memory pool — with odd and often undesirable consequences[3].

Assuming that framebuffer memory originates and is consumed within '`vga.fb`', we have an occam-π process with the structure:

```
PROC vga.fb (CT.DRV? link)
  CT.GUI.FB! fb.cli:
  CT.GUI.FB? fb.svr:
```

[3]Mapping process memory (typically a process's *workspace*) into video memory, or vice-versa, does provide an interesting way of visualising process behaviour in RMoX, however.

```
  SEQ
    fb.cli, fb.svr := MOBILE CT.GUI.FB            -- create channel-bundle

    ...  other initialisation and declarations

    PAR
      WHILE TRUE
        link[in] ? CASE
          CT.DRV.R! ret:
          open.device; ret                        -- request to open device
            IF
              DEFINED fb.cli
                ret[out] ! device; fb.cli         -- return bundle client-end
              TRUE
                ret[out] ! device.busy
          ...  other cases

      PLACED MOBILE []BYTE framebuffer AT ...:
      WHILE TRUE
        fb.svr[in] ? CASE                         -- request from connected client
          get.buffer
            fb.svr[out] ! buffer; framebuffer     -- outgoing framebuffer
          put.buffer; framebuffer                 -- incoming framebuffer
            SKIP
:
```

That has the mobility model:

$$mobility\ \mathrm{VFB} = \left\{\langle \overline{link}?^r, r!^a\rangle, \langle \bar{a}_{[1]}!^b\rangle, \langle \bar{a}_{[0]}?^c\rangle\right\}$$

The escape information here indicates that mobiles are generated and consumed at the server-end of the channel bundle $\bar{a}$, whilst the client-end of this bundle, a, escapes through another channel bundle r that the process receives from its $\overline{link}$ parameter.

Instead of going into detail for the other processes involved, that would require a significant amount of space, the generic forwarding and use of connections is considered.

4.2.1. Client Processes

The mechanism by which dynamic connections to device-drivers and suchlike are established involves sending the client-end of a *return* channel-bundle along with the request. A client process (e.g. '`application`' from figure 3) therefore typically has the structure:

```
PROC client (SHARED CT.DRV! to.drv)
  CT.DRV.R! r.cli:
  CT.DRV.R? r.svr:
  CT.GUI.FB! guilink:
  SEQ
    r.cli, r.svr := MOBILE CT.DRV.R               -- create response channel-bundle

    CLAIM to.drv
      to.drv[in] ! open.device; r.cli             -- send request

    r.svr[out] ? CASE                             -- wait for response
      device.busy
        ...  fail gracefully
      device; guilink
        ...  use 'guilink'
:
```

This has the mobility model:

$$mobility\ \mathrm{CLI} = \left\{\langle to.drv!^{e}\rangle, \langle \bar{e}?^{f}\rangle\right\} \cup \mathbb{M}$$

where $\mathbb{M}$ is the mobility model for the part of the process that uses the '`guilink`' connection to the underlying service, and will communicate directly on the individual channels within f.

Connecting this client and the '`vga.fb`' processes directly, with renaming for parameter passing, gives the following mobility set:

$$\left\{\langle \bar{A}?^{r}, r!^{a}\rangle, \langle \bar{a}_{[1]}!^{b}\rangle, \langle \bar{a}_{[0]}?^{c}\rangle, \langle A!^{e}\rangle, \langle \bar{e}?^{f}\rangle\right\} \cup \mathbb{M}$$

Hiding the internal link $A, \bar{A}$ gives:

$$\left\{\langle e!^{a}\rangle, \langle \bar{a}_{[1]}!^{b}\rangle, \langle \bar{a}_{[0]}?^{c}\rangle, \langle \bar{e}?^{f}\rangle\right\} \cup \mathbb{M}$$

If we take a well-behaved client implementation for $\mathbb{M}$ — i.e. one that inputs a mobile (framebuffer) from the underlying driver, modifies it in some way and then returns it, without destroying or creating these ($\mathbb{M} = \{\langle f_{[1]}?^{x}, f_{[0]}!^{x}\rangle\}$) — we get:

$$\left\{\langle e!^{a}\rangle, \langle \bar{a}_{[1]}!^{b}\rangle, \langle \bar{a}_{[0]}?^{c}\rangle, \langle \bar{e}?^{f}\rangle, \langle f_{[1]}?^{x}, f_{[0]}!^{x}\rangle\right\}$$

Subsequently hiding e, which represents the '`CT.DRV.R`' link, causes f to be renamed to a, giving the set:

$$\left\{\langle \bar{a}_{[1]}!^{b}\rangle, \langle \bar{a}_{[0]}?^{c}\rangle, \langle a_{[1]}?^{x}, a_{[0]}!^{x}\rangle\right\}$$

Logically speaking, and for this closed system, b and c must represent the same thing — in this case, mobile framebuffers. Thus we have a guarantee that mobiles generated within the '`vga.fb`' process are returned there, for this small system.

On the other hand, a less well-behaved client implementation for $\mathbb{M}$ could be one that occasionally *loses* one of the framebuffers received, instead of returning it (i.e. $\mathbb{M} = \{\langle f_{[1]}?^{x}, f_{[0]}!^{x}\rangle, \langle f_{[1]}?^{y}\rangle\}$). This ultimately gives the mobility set:

$$\left\{\langle \bar{a}_{[1]}!^{b}\rangle, \langle \bar{a}_{[0]}?^{c}\rangle, \langle a_{[1]}?^{x}, a_{[0]}!^{x}\rangle, \langle a_{[1]}?^{y}\rangle\right\}$$

As before, b and c must represent the same mobiles, so the only mobiles received back must have been those sent. However, the presence of the sequence $\langle a_{[1]}?^{y}\rangle$ indicates that framebuffers can be received and then discarded by this client.

Another badly behaved client implementation is one that generates mobiles and returns these as framebuffers, in addition to the normal behaviour, e.g. $\mathbb{M} = \{\langle f_{[1]}?^{x}, f_{[0]}!^{x}\rangle, \langle f_{[0]}!^{z}\rangle\}$. This gives the resulting mobility set:

$$\left\{\langle \bar{a}_{[1]}!^{b}\rangle, \langle \bar{a}_{[0]}?^{c}\rangle, \langle a_{[1]}?^{x}, a_{[0]}!^{x}\rangle, \langle a_{[0]}!^{z}\rangle\right\}$$

In this case, b and c do not necessarily represent the same mobiles — as while x can only be b, c can be either x (and therefore b) or z. Thus there is the possibility that mobiles are returned to the '`vga.fb`' driver that did not originate there.

4.2.2. Infrastructure

Within RMoX, such client and server processes are normally connected through a network of processes that route requests around the system. From figure 3, this includes the '`driver.core`', '`service.core`' and '`kernel`' processes.

In earlier versions of RMoX [19], both requests and their responses were routed through the infrastructure. This is no longer the case — requests now include, as part of the request,

a mobile channel-end that is used for the response. This is a cleaner approach in many respects and is more efficient in most cases. From the client's perspective, a little more work is involved when establishing connections, since the return channel-bundle must be allocated. Most of the infrastructure components within RMoX consist of a single *server-end* channel-bundle on which requests are received, whose *client-end* is shared between multiple processes, and multiple *client-ends* connecting to other server processes such as '`vga.fb`' and other infrastructure components.

A very general implementation of an infrastructure component is:

```
PROC route (CT.DRV? in, CT.DRV! out.this, SHARED CT.DRV! out.next)
  WHILE TRUE
    in[in] ? CASE
      CT.DRV.R! ret:
      open.device; ret
        IF
          request.for.this
            out.this[in] ! open.device; ret
          NOT invalid
            CLAIM out.next!
              out.next[in] ! open.device; ret
          TRUE
            ret[out] ! no.such.device
      ...  other cases
:
```

The mobility model of this process is:

$$mobility\ \mathrm{Rt} = \left\{\langle \overline{in}?^{a}, out.this!^{a}\rangle, \langle \overline{in}?^{b}, out.next!^{b}\rangle, \langle \overline{in}?^{c}\rangle\right\}$$

The last component indicates that this routing process may discard the request (and the response channel-end) internally — after it has reported an error back on the response channel, of course.

With the '`route`' process as it is, there would need to be an additional process at the end of this *chain* that responds to all connection requests with an error, e.g.:

```
PROC end.route (CT.DRV? in)
  WHILE TRUE
    in[in] ? CASE
      CT.DRV.R! ret:
      open.device; ret
        ret[out] ! no.such.device
      ...  other cases
:
```

$$mobility\ \mathrm{ERt} = \left\{\langle \overline{in}?^{x}\rangle\right\}$$

Combining one '`route`' process and one '`end.route`' process with the existing '`vga.fb`' and '`client`' processes produces the network shown in figure 4.
This has the following mobility model:

$$\left\{\langle \bar{C}?^{r}, r!^{a}\rangle, \langle \bar{a}_{[1]}!^{b}\rangle, \langle \bar{a}_{[0]}?^{c}\rangle, \langle A!^{e}\rangle, \langle \bar{e}?^{f}\rangle, \langle \bar{B}?^{x}\rangle, \langle \bar{A}?^{a}, C!^{a}\rangle, \langle \bar{A}?^{b}, B!^{b}\rangle, \langle \bar{A}?^{c}\rangle\right\} \cup \mathbb{M}$$

Hiding the internal links A, B and C gives:

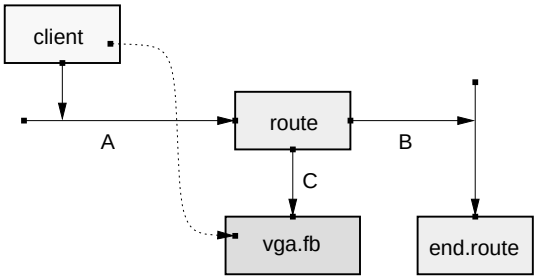


Figure 4. RMoX routing infrastructure.

$$\xrightarrow{\setminus\{A\}} \left\{\langle \bar{C}?^r, r!^a\rangle, \langle \bar{a}_{[1]}!^b\rangle, \langle \bar{a}_{[0]}?^c\rangle, \langle \bar{e}?^f\rangle, \langle \bar{B}?^x\rangle, \langle C!^e\rangle, \langle B!^e\rangle\right\} \cup \mathbb{M}$$

$$\xrightarrow{\setminus\{B\}} \left\{\langle \bar{C}?^r, r!^a\rangle, \langle \bar{a}_{[1]}!^b\rangle, \langle \bar{a}_{[0]}?^c\rangle, \langle \bar{e}?^f\rangle, \langle C!^e\rangle\right\} \cup \mathbb{M}$$

$$\xrightarrow{\setminus\{C\}} \left\{\langle e!^a\rangle, \langle \bar{a}_{[1]}!^b\rangle, \langle \bar{a}_{[0]}?^c\rangle, \langle \bar{e}?^f\rangle\right\} \cup \mathbb{M}$$

This system has an identical mobile escape model to the earlier directly connected '`client`' and '`vga.fb`' system. As such, the system can still be sure that framebuffer mobiles generated by '`vga.fb`' are returned there.

5. Related Research

The use of *escape analysis* for determining various properties of *dynamic* systems stems from the functional programming community. One use here is for determining which parts of an expression *escape* a particular function, and if they can therefore be allocated on the stack (i.e. they are local to the function) [20]. More recently, escape analysis has been used in conjunction with *object-oriented* languages, such as Java [21]. Here it can be used to determine the boundaries of object references within the object graph, for the purposes of stack allocation and other garbage collector (GC) optimisations [22]. With the increasing use of *multi-core* and *multi-processor* systems, this type of analysis is also used to discover which objects are local to which threads (known as *thread escape analysis*), allowing a variety of optimisations [23].

While escape analysis for functional languages is generally well-understood, it gets extremely complex for object-oriented languages such as C++ and Java. Features inherent to object-oriented languages, *inheritance* and *polymorphism* in particular, have a significant impact on formal reasoning. The number of objects typically involved also create problems for automated analysis (state-space explosion).

The escape analysis described here is more straightforward, but is sufficient for determining the particular properties identified earlier. The compositional nature of occam-π and CSP helps significantly, allowing analysis to be done in a divide-and-conquer manner, or to enable analysis to be performed on a subset of processes within a system (as shown in section 4.2.2).

6. Conclusions and Future Work

This paper has presented a straightforward technique for *mobile escape analysis* in occam-π, and its application to various kinds of process network. The analysis provides for the checking of particular design-time properties of a system and can permit certain optimisations in the implementation. At the top-level of a system, this escape analysis can also provide hints towards efficient distribution of the system across multiple nodes — by identifying those parts

interconnected through mobile communication (and whose efficiency of implementation is greatly increased with shared-memory). Although the work here has focused on occam-π, the techniques are applicable to other process-oriented languages and frameworks.

The semantic model for $mobility$ presented here is not quite complete. Some of the formal rules for process composition have yet to be specified, though we have a good informal understanding of their operation. Another aspect yet to be fully considered is one of *mobile processes*. These can contain other mobiles as part of their state (within local variables), and as such warrant special treatment. The analysis techniques shown provide a very general model for mobile processes — in practice this either results in a larger state-space (where mobiles within mobile processes are tracked individually), or a loss in accuracy (e.g. treating a mobile process as $CHAOS$). Once a complete semantic model has been established, it can be checked for validity, and the concept of *mobility refinement* investigated thoroughly.

For the practical application of this work, the existing occam-π compiler needs to be modified to analyse and generate machine readable representations of mobile escape. Some portion of this work is already in place, discussed briefly in [24], where the compiler has been extended to generate CSP style behavioural models (in XML) of individual PROCs occam-π code. The mobile escape information obtained will be included within these XML models, incorporating attributes such as type. A separate but not overly complex tool will be required to manipulate and check particular properties of these — e.g. that an application process does not discard or generate framebuffer mobiles (section 4.2). How such information can be recorded and put to use for compiler and run-time optimisations is an issue for future work.

Acknowledgements

This work was funded by EPSRC grant EP/D061822/1. The author would like to thank the anonymous reviewers for their input on an earlier version of this work.

References

[1] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[2] Inmos Limited. occam 2.1 Reference Manual. Technical report, Inmos Limited, May 1995. Available at: `http://wotug.org/occam/`.

[3] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-13-153271-5.

[4] M.H. Goldsmith, A.W. Roscoe, and B.G.O. Scott. Denotational Semantics for occam2, Part 1. In *Transputer Communications*, volume 1 (2), pages 65–91. Wiley and Sons Ltd., UK, November 1993.

[5] M.H. Goldsmith, A.W. Roscoe, and B.G.O. Scott. Denotational Semantics for occam2, Part 2. In *Transputer Communications*, volume 2 (1), pages 25–67. Wiley and Sons Ltd., UK, March 1994.

[6] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN: 0-52165-869-1.

[7] P. Andrews, A. Sampson, J. Bjørndalen, S. Stepney, J. Timmis, D. Warren, and P. Welch. Investigating patterns for the process-oriented modelling and simulation of space in complex systems. In S. Bullock, J. Noble, R. Watson, and M. A. Bedau, editors, *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*, pages 17–24. MIT Press, Cambridge, MA, 2008.

[8] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent, June 2003.

[9] P.H. Welch and F.R.M. Barnes. Mobile Barriers for occam-pi: Semantics, Implementation and Application. In J.F. Broenink, H.W. Roebbers, J.P.E. Sunter, P.H. Welch, and D.C. Wood, editors, *Communicating Process Architectures 2005*, volume 63 of *Concurrent Systems Engineering Series*, pages 289–316, Amsterdam, The Netherlands, September 2005. IOS Press. ISBN: 1-58603-561-4.

[10] P.H. Welch and F.R.M. Barnes. A CSP model for mobile channels. In *Proceedings of Communicating Process Architectures 2008*. IOS Press, September 2008.

[11] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0-13-674409-5.

[12] F.R.M. Barnes, P.H. Welch, and A.T. Sampson. Barrier synchronisations for occam-pi. In Hamid R. Arabnia, editor, *Proceedings of PDPTA 2005*, pages 173–179, Las Vegas, Nevada, USA, June 2005. CSREA press.

[13] 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, *Proceedings of Communicating Process Architectures 2001*. IOS Press, September 2001.

[14] P.H. Welch, F.R.M. Barnes, and F.A.C. Polack. Communicating Complex Systems. In Michael G. Hinchey, editor, *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS-2006)*, pages 107–117, Stanford, California, August 2006. IEEE. ISBN: 0-7695-2530-X.

[15] F.R.M. Barnes and P.H. Welch. Prioritised dynamic communicating and mobile processes. *IEE Proceedings – Software*, 150(2):121–136, April 2003.

[16] J.C.P. Woodcock and A.L.C. Cavalcanti. The Semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.

[17] S. Schneider and H. Treharne. Communicating B Machines. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 251–258. Springer-Verlag, January 2002.

[18] F.R.M. Barnes, C.L. Jacobsen, and B. Vinter. RMoX: a Raw Metal occam Experiment. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 269–288, Amsterdam, The Netherlands, September 2003. IOS Press. ISBN: 1-58603-381-6.

[19] Carl G. Ritson and Frederick R.M. Barnes. A Process Oriented Approach to USB Driver Development. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering Series*, pages 323–338, Amsterdam, The Netherlands, July 2007. IOS Press. ISBN: 978-1-58603-767-3.

[20] Young Gill Park and Benjamin Goldberg. Higher order escape analysis: Optimizing stack allocation in functional program implementations. In *Proceedings of ESOP '90*, volume 432 of *LNCS*, pages 152–160. Springer-Verlag, 1990.

[21] B. Joy, J. Gosling, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996. ISBN: 0-20-163451-1.

[22] Bruno Blanchet. Escape analysis for Java(TM): Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, 2003.

[23] Kyungwoo Lee, Xing Fang, and Samuel P. Midkiff. Practical escape analyses: how good are they? In *Proceedings of VEE '07*, pages 180–190. ACM, 2007.

[24] Frederick R. M. Barnes and Carl G. Ritson. Checking process-oriented operating system behaviour using CSP and refinement. In *PLOS 2009*. ACM. To Appear.