Toward Robust Software Systems

Matej Košík
Slovak University of Technology
kosik@fiit.stuba.sk

Jiří Šafařík
University of West Bohemia
safarikj@kiv.zcu.cz

Abstract

Kernel space represents unforgiving but lucid environment. It is a great environment where we can compare various competing language-based approaches for improving robustness of software systems. We can compare elegance of chosen security models, the size of the trusted computing base, performance penalty (or boost). We have decided to take object-capability security model and set it to work for ensuring robustness of the kernel with the hope to observe and compare various relevant software qualities. We describe relationship between programming languages and object-capability security model. We identify limitations of this approach and propose a solution that overcomes those limitations.

Categories and Subject Descriptors F.3.2 [*Logics and meanings of programs*]: Semantics of programming languages—Process models

General Terms Design, Experimentation, Languages, Reliability

Keywords denial of service attacks, memory management, pi-calculus, asynchronous message-passing

1. Introduction

Currently, the society depends on inherently fragile software systems. These systems are:

- as reliable as their least reliable component,
- as secure as their most severe security flaw,
- as safe as their most malicious component.

This fragility can be recognized at various levels:

- operating system kernel,
- whole user-space desktop,
- individual user-space software system.

[Copyright notice will appear here once 'preprint' option is removed.]

A bug or a malicious code in any driver can have arbitrary effect on the whole system. A bug or a malicious code, in any application we run, can have arbitrary effect on the whole user's account. A bug or a malicious code in any component of an application can have arbitrary effect on the whole application or the whole user's account.

Concerning building robust user-space software systems, we can rely on the *object-capability security model* [10] directly supported by *object-capability programming languages*. Then, our effort to enforce security policies over untrusted subsystems can rely on the following axioms:

- Capabilities to objects[1] are not forgeable.
- Subjects can work only with those objects to which they have a capability.
- A subject can have or get a capability to an object only by: *initial conditions*, *parenthood*, *introduction* and *endowment* as stated by Miller [10].

Often, when some software system starts, some subjects inherently have access to other objects. This phenomenon is called *connectivity by initial conditions*. It can be usually inferred directly from the source code at compile time simply taking into consideration statical scoping rules. If one subject creates an object, it is automatically given a capability to it. This phenomenon is called *connectivity by parenthood*. It is closely related to calling constructors of objects. *Connectivity by introduction* refers to a possibility to pass capability to an object in a message to other object. *Connectivity by endowment* is related to abstractions as they are understood in the context of the lambda-calculus or the pi-calculus. Abstractions can refer (with their free variables) to any object that is visible in the context where we literally construct them and abstractions will continue to refer to those objects regardless of the position from which we ultimately invoke them.

Object-capability programming languages enable us to create robust (user-space or kernel-space) software systems. Various experimental programming languages support this kind of software composition: E [10] (an object-capability

[1] The terms *subject* and *object* are here used in concord with the standard security literature. *Subjects* are active entities that, during their lifetime, may try to perform various (supported) *operations* with *objects*. These *operations* are allowed or denied with respect to *permissions* a given *subject* possesses.

language by design), Joe-E [9] (retrofitted Java), Emily [18] (retrofitted Ocaml), Caja (retrofitted JavaScript [1]). Here, by retrofitting we mean making sure that there are no ways how could one violate the object-capability security model[2].

There are various degrees of robustness [10]. The two most interesting degrees of robustness are:

- defensive consistency (weaker form of robustness),
- defensive correctness (stronger form of robustness).

The first form of robustness is directly connected with existing object-capability languages because they support it by design. The stronger form of robustness is envisioned, but no programming language actually supports it yet.

In the paper, we describe differences between these two forms of robustness—why *defensive correctness* is more desirable than mere *defensive consistency*. We demonstrate what kind of obvious problems are not addressed by current object-capability programming languages. In a context of a chosen object-capability programming language we propose a solution.

2. Defensively consistent processes

We say that a given algorithm is *partially correct* if it never returns incorrect result. It is possible to generalize this property also for processes. We can say that a given process[3] is *defensively consistent* [10] if it never interacts with its environment in an incorrect way.

There are platforms where it is possible to determine possible interaction of processes. One of them is pi-calculus. Each process can interact with other processes only by message-passing over channels designated by its free variables. Then, knowing something about channels used by a process, we can determine how a given process can influence its environment. This is called *authority*[4] of this process.

Frequently, we do not prove that given untrusted process is defensively consistent; we simply conjecture that it is. This conjecture may be supported by code review, testing, non-existence of counter-examples, non-existence of successful attacks despite our public promise to pay bounties for them. This risk may still be completely acceptable if we follow *principle of least authority (POLA)*. Then, even if our conjecture were false, the untrusted process would not be able to cause more than acceptable and/or unavoidable amount of damage.

3. Security audit of an untrusted process

Figure 1 shows a structure of a very simple kernel-space software system composed from *trusted* and *untrusted* processes. Trusted processes are allowed to inline arbitrary C code. Therefore, they belong to the trusted computing base. Untrusted processes are not allowed to inline arbitrary C code. Untrusted processes are outside trusted computing base because authority of these processes—how can they interact with the environment—can be easily determined. Every untrusted process can interact with its environment only via channels designated by its free variables. If we consider values of these free variables in the original context, we can determine what channels interconnect given process with its context. Taking into account other processes interacting on those channels, we can determine how can our process affect its environment.

For example, the `Timer` process in Figure 1 can receive messages from two channels and send messages to three other channels. Let us focus on those three channels to which it can send messages. With respect to the behavior of processes that listen to those channels, the `Timer` process has the authority:

- to add itself as an observer of IRQ 1
- to write any byte to I/O port `0x40`
- to write any byte to I/O port `0x43`

This enables us to make a qualified decision whether we are willing to run a given untrusted code with this authority or not. Processes that require *excess authority* can be rejected. In case of platforms which enable programmers to obey principle of least authority without functional compromises, excess authority of untrusted processes is unacceptable.

4. Defensively correct processes

We say that a given algorithm is *totally correct* if:

- it is partially correct (safety property)
- and if it terminates (liveness property)

It is possible to generalize this property also for processes. We can say that a given process is *defensively correct* [10] if:

- it is defensively consistent (safety property)
- and it is *responsive*[5] (liveness property).

If we understand what is defensive correctness, we can often notice that the platform one has chosen is incapable of supporting it. Sometimes one can improve it. This has already happened a few times mostly with platforms that support synchronous message-passing. For example Minix developers realized that they must change the IPC primitives [2].

There are at least two reasonable ways how we can build a robust operating system kernel:

- we can adhere to microkernel architecture
- or we can take advantage of programming languages.

[2] Capability community sometimes refers to this process as *taming*.

[3] A process can play a role of a server in a client-server system or a peer in a peer-to-peer system.

[4] Despite the fact that mainstream uses the term *privilege*, we use the term *authority*. The former term is not defined sufficiently clearly and correctly to be useful. Consequently, *principle of least privilege* (POLP) is useless too.

[5] We say that a subsystem is *responsive* if it is not possible to write a client that can force given process to stop providing services for the other well-behaving clients.

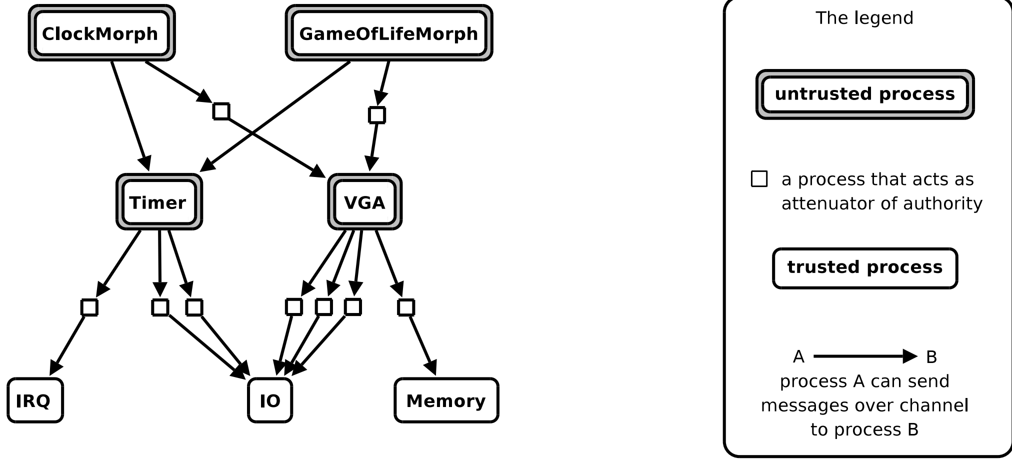

Figure 1. An example of a software system composed from multiple processes. Each of these processes follows *principle of least authority*. Untrusted processes are started in a sandbox and we judiciously choose what kind of additional capabilities we send them. This is determined at design time by informed designer. They can be reviewed by trusted auditor. An uninformed user is not expected to make any such decisions himself. All the untrusted processes thus have sufficient authority to be able to provide expected services. Excess authority is avoided by creation of proxy processes that are placed between clients and servers. They appropriately confine the original powerful services to a level that is sufficient for a given client.

In the first case, we can use any programming language and we can take advantages of security mechanisms provided by microprocessor designers. In the second case, we dismiss security mechanisms provided by microprocessor designers and we rely on security mechanisms provided by the programming language. The quality of instances of these two approaches can be commonly measured also by the size of the *trusted computing base (TCB)*. Minix [19] represents an instance of the first approach. Its trusted computing base is at least 17 000 lines of code[6]. Singularity [3] represents the second approach. Its trusted computing base has at least 150 000 lines of code[7]

A good choice for building robust operating system is the Pict programming language [11]. Like Java and C#, it provides *memory safety*[8] that eliminates certain classes of errors in untrusted code. Memory safety is provided by Pict at a lower price:

- its runtime fits in 800 lines of C code
- its compiler fits in 7000 lines of Ocaml code

This was achieved by choosing reasonable *core language* and defining higher level language constructs as a *syntactic sugar* that is mapped to the core language. Several concepts that are, in different languages, separate are here expressed as a different combination of the same set of orthogonal concepts available in the core language. This simplifies the compiler as well as the runtime of the language. Syntactic sugar does not make the language more expressive but it makes it easier to use.

In addition to plain *memory safety*, Pict programming language can be easily retrofitted [5] to provide much more interesting feature—the *object-capability security model* partially sketched in Section 1. It directly enables us to confine authority of untrusted processes to a level at which they can still provide expected services but they cannot interact with their environment in an undesired way. In short, we are able to build defensively consistent software systems with a small trusted computing base.

Unfortunately, this is only partially satisfactory. Neither Pict, nor any other object-capability programming language alone, enables us to build defensively correct software systems. We must modestly admit that these languages are too weak. For Pict, we demonstrate these weaknesses in the next section. In the rest of the paper, we propose essential and sufficient changes in the syntax and the semantics of the Pict programming language that will enable us to use it for building defensively correct software systems.

[6] It is formed at least by the microkernel, the process manager and libraries linked with the process manager.

[7] It is formed by hardware abstraction layer, memory manager, metadata manager, loader, scheduler, channel manager, I/O manager, security manager, MSIL code translators, see Figure 2 in [4].

[8] *Memory safety* is sometimes confusingly referred to as *managed code*.

5. Weaknesses of the Pict programming language

Denial-of-service attacks in this section demonstrate certain inherent weaknesses of the Pict programming language. We demonstrate these weaknesses in a context of a simple operating system whose structure is shown in Figure 1.

There is a dissonance between abstract definition of the Pict programming language described in [14] and the actual implementation of the compiler and the runtime of this language. According to the abstract definition, process terms can grow to arbitrary size. It is not possible to represent such terms in computers with bounded memory. Thus, it is not surprising that the abstract definition of the language faithfully describes behavior of real systems only up to the point where real systems fit into the available memory. At the point when no more reductions can be performed due to lack of memory, the real implementation must deviate from the abstract definition. In our case, the Pict runtime does the most intelligent action it can in this case do: it prints relevant runtime error message and it shuts the whole system down.

If all untrusted processes can grow to arbitrary size then all untrusted processes have the authority to shut the whole system down. This is in conflict with our goal to use Pict programming language for building defensively correct software systems.

5.1 A denial-of-service (DoS) attack: a wabbit

If some process acts as a wabbit:

```
def wabbit [] = ( wabbit![] | wabbit![] )
run wabbit![]
```

then it quickly exhausts all the available free memory. The runtime can trivially recognize, that the system as a whole needs more memory then it is currently available in order to perform intended reduction. However, the runtime is unable to determine which particular subsystem uses excess memory because it is unable to recognize these subsystems.

The platform should thus enable programmer to put every untrusted process to a separate *memory domain* with its own memory quota. It is up to the programmer to judiciously create one domain for each untrusted process. The platform will then ensure fair memory accounting. That is, if some subject from domain creates some new object, then the newly created object will belong to the same domain as its creator. If this rule holds transitively, offending memory domains will manifest themselves by exceeding their memory quota. These kinds of attacks can be trivially identified and all the objects belonging to the corresponding domain can be deleted. This may disable some functionality. We can later investigate precise reasons why given domain exceeds memory and fix that. In the meantime, unrelated processes will be able to continue.

5.2 A DoS attack: a spammer

We say that a given domain behaves as a spammer if it sends excess messages to a channel ch owned by another domain.

```
def spammer [] = ( ch![] | spammer![] )
run spammer![]
```

These messages, until they are processed, must be stored somewhere. Spammers can prevent progress by depleting available free space. To be able to efficiently defend the rest of the system from spammer-like behaviors we must update the platform to ensure that sent messages are accounted to (the domain of) senders. That is, we must change the semantics of the message-send operations.

As a result, if some domain contains a spammer process, it will then manifest itself by exceeding its memory quota. The runtime will be able to recognize this and delete such domain. This will enable progress of other processes if they are independent from the deleted domain.

5.3 A DoS attack: a hostile client

Suppose that we ubiquitously change the semantics of the messagesend as we indicated in the previous section—i.e. sent messages are always accounted to the sender. According to Figure 1, `ClockMorph` can invoke `Timer`s services. `Timer` enables its clients to schedule periodic sending of messages to a designated channel with a specified period. All such messages autonomously sent by `Timer` will be accounted to the `Timer`. The Timer is thus vulnerable to its clients because if some hostile client decides not to consume those sent messages then it can cause Timer to exceed its memory limit.

This vulnerability can be straightforwardly eliminated with two variants of the *send* operation:

- *regular send* operation where the sent message is accounted to the sender,
- and so called *donating send* operation where the sent message is accounted to the receiver.

With these two kinds of send operations we have a complete set of mechanisms required for building defensively correct software systems. Clients will be given the permission to invoke services of servers via *regular sends*. If servers are expected to autonomously contact clients, then clients will be obliged[9] to grant servers the permission to perform *donating* send to the some designated channel.

These attacks played a key role during formulating exact semantics of the updated version of the programming language. By the claim: “given language enables us to achieve defensive correctness” we meant that we have considered various scenarios:

- what services servers might want to provide,

[9] There are ways how to check fulfillment of this obligation both, statically at compile-time as well as dynamically at run-time.

- what could clients try to do,

and whether in given case we would have enough mechanisms to defend server from any kind of client. These and other attacks are listed in Section 10.4.9 (Discussion) in the documentation [7].

5.4 Additional mechanisms

If a given platform supports the object-capability security model introduced in Section 1, it can be used for building defensively consistent software systems. If we want to build defensively correct software systems, we must extend the original object-capability security model in the following way:

- We should be able to start untrusted subsystems in separate *domains* and it should be possible to determine the amount of memory consumed by each *domain*.
- For message-passing, two kinds of *send* operations are necessary. In case of the *regular send*, the sent message is accounted to the sender. In case of the *donating send*, the sent message is accounted to the receiver.

We address defensive correctness in the presence of asynchronous message-passing because it is the most expressive means of communication. With asynchronous message-passing (over channels) we can model all the other types of communications: function calls, procedure calls, rendezvous between two concurrent processes etc.

We ignore the problems related to deadlocks because they are well covered elsewhere. In our experimental kernels [6] we avoid deadlocks in a similar way how they are avoided in Minix [19]. Clients and servers form partial order which defines allowed invocation of services via synchronous rendezvous. Asynchronous message-passing is allowed in any direction. This approach is strong enough to avoid deadlock and liberal enough to enable us to write a complete operating system kernel. If we needed true peer-to-peer architecture we would probably have employ *event-loop concurrency* [10].

6. The original programming language

Declared [13] primary motivation of the Pict project was to "design and implement a high-level concurrent programming language purely in terms of the pi-calculus primitives. A number of programming language designs have combined pi-calculus-like constructs with a functional core language, but none have gone so far as to take communication as a sole means of computation." The Pict language definition [14] describes syntax of the language, extralinguistic features, semantics of the core language, expansion of the syntactic sugar constructs into the core language and the type system. The semantics of the core language is defined in Section 13.4 by a small set of reduction rules. They are useful when we want to understand the semantics of the core language because they are concise and precise. As they are stated, they cannot be directly efficiently implemented because:

Proc	::=	`()`
	|	*id* `!` *Val*
	|	*id* `$` *Val*
	|	id_1 `?` id_2 `=` *Proc*
	|	id_1 `?*` id_2 `=` *Proc*
	|	`(` $Proc_1$ `|` $Proc_2$ `)`
	|	`(` `new` *id* `:` *Type Proc* `)`
	|	`(lim` *integer Proc*`)`
	|	`(ccode` *integer string*`)`
Value	::=	`[]`
	|	*id*
Type	::=	`[]`
	|	`P.`*Type* where $P \subseteq \{$`?`, `!`, `$`$\}$

Figure 2. Syntax of the core language.

- they rely on the human creativity to find out proper structurally congruent terms in order to enable reduction
- and they rely on operations such as variable substitution in process terms that cannot be efficiently implemented.

To fix this, Turner in his thesis [20] gradually refines the original set of reduction rules in several steps to a final set of reduction rules that is simulated by the original set of reduction rules and which can be efficiently implemented. The proposed abstract machine and reduction rules are described in Section 7.11 in his thesis. Pierce and Turner implemented a compiler and a runtime for this language. Reorganization [5] of its standard library [12] and few additional simple restrictions enable us to use this language for building defensively consistent software systems [6, 8]. Section 5 demonstrates weaknesses of the original language. It demonstrates that the original language cannot be used for creating defensively correct software systems. We consider particular attacks described in Section 5 in this paper as well as various other attacks we have considered elsewhere (Section 10.4.8 in [7]) as a motivation for the revised version of the language.

7. Revised programming language

Updated definition of the Pict programming language cannot fit in this document but we can describe semantics of the proposed additions on a subset of the original Pict programming language. The chosen subset of the Pict programming language is expressive enough to enable us to model original problems we are trying to address with new language constructs that we add to the language.

7.1 Grammar

Syntax of the language is captured in Figure 2.

Null process `()` has no observable behavior.

Output process *id* `!` *Val* writes a given value *Val* to a channel designated by capability bound in variable *id*. Here *id* is a token class of all the legal variable names (identifiers) such `x`, `y`, `z` etc.

Donor output process *id* `$` *Val* writes a given value *Val* to a channel designated by capability bound in variable *id*. Values queued to channels with ordinary output action are accounted to senders. Values queued to channels with donor output action are accounted to owner of the designated channel.

Input process id_1 `?` id_2 `=` *Proc* reads a value from a channel designated by capability bound in variable id_1. Received value is bound to variable id_2. If no value is currently available in the channel, input process blocks until some is available. Then it behaves as a given continuation *Proc*.

One replicated input process id_1 `?*` id_2 `=` *Proc* is equivalent to infinite number of ordinary input processes composed in parallel. This primitive construct can be used to define meaning of a convenient but semantically complicated `def` construct available in the original Pict programming language.

Composition of processes `(` $Proc_1$ `|` $Proc_2$ `)` enables process $Proc_1$ and $Proc_2$ to communicate via channels to which they both have access because they hold appropriate capabilities.

`( new` *id* `:` *Type Proc* `)` creates a new channel of a given *Type* and binds capability designating this new channel to variable *id*. The whole process then behaves as a given continuation *Proc*.

`(lim` *integer Proc*`)` will execute process *Proc* in a new trust domain with a specified memory quota. It may for example mean: number of 32-bit words a given domain is allowed to allocate from the common heap shared by all domains.

Behavior of the primitive process `(ccode` *integer string*`)` is determined by inlined C code delimited by two `@` signs. In practice, this construct is essential to enable interaction of the system with its environment (e.g. different regions of memory, various I/O ports etc.). However, the `ccode` construct should not be available in untrusted processes because it would enable them to circumvent the object-capability security model.

All literals and all variables in our language have a type. We support only single basic type: *unit type* denoted as `[]`. It has a single value: *unit value* denoted as `[]`. They are denoted in the same way but from the context we can always disambiguate them.

Our *channel types* are constructed from two components: chosen permission rights *P* that determine which of the three operations with a capability are allowed and *Type* that determines the type of values that can be sent to and/or received from the designated channel. *Type* can be either *unit type* or again some *channel type*. Typing and subtyping rules can be specified rigorously, but in this text we describe them only very briefly. Our type system is an extension of `i/o` types defined in [15]. In addition to *input* and *output* permissions we recognize also *donor output* permission. Very shortly, if we have a capability `x` of type P.*Type* then:

- a subject can perform the input operation on `x` only if `?` $\in$ P
- a subject can perform the output operation on `x` only if `!` $\in$ P
- a subject can perform the donor output operation on `x` only if `$` $\in$ P

It is often very useful to be able to drop various permissions granted by a capability before handing it over to an untrusted party. This, attenuation of permissions, is enabled by subtyping rules shown in Figure 3. The type of any value can be freely promoted to any of its supertypes. New capabilities created by promotion designate the same channel but enable their holders to perform only a subset of the original operations.

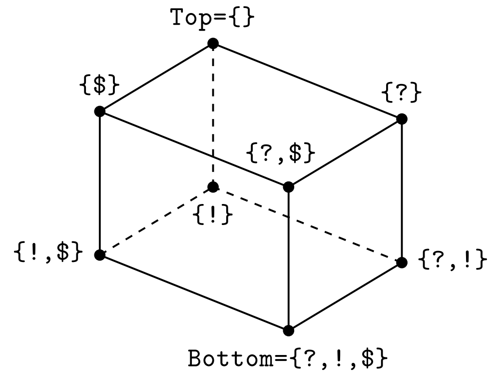

Figure 3. Subtyping on the surface relationship defined via Hasse's diagram.

7.2 Free variables

The **fv**(p) function returns the set of all free variables of a given process term p. This function is used by our reduction rules. We provide recursive definition over abstract syntax tree.

$$
\begin{aligned}
\mathbf{fv}(\texttt{()}) &\stackrel{\text{def}}{=} \varnothing \\
\mathbf{fv}(\texttt{(}p_1\texttt{|}p_2\texttt{)}) &\stackrel{\text{def}}{=} \mathbf{fv}(p_1) \cup \mathbf{fv}(p_2) \\
\mathbf{fv}(\texttt{id!v}) &\stackrel{\text{def}}{=} \{\texttt{id}\} \cup \mathbf{fvv}(\texttt{v}) \\
\mathbf{fv}(\texttt{id\$v}) &\stackrel{\text{def}}{=} \mathbf{fv}(\texttt{id!v}) \\
\mathbf{fv}(\texttt{id}_1\texttt{?id}_2\texttt{=p}) &\stackrel{\text{def}}{=} \{\texttt{id}_1\} \cup (\mathbf{fv}(\texttt{p}) \setminus \{\texttt{id}_2\}) \\
\mathbf{fv}(\texttt{id}_1\texttt{?*id}_2\texttt{=p}) &\stackrel{\text{def}}{=} \mathbf{fv}(\texttt{id}_1\texttt{?id}_2\texttt{=p}) \\
\mathbf{fv}(\texttt{(new id:t p)}) &\stackrel{\text{def}}{=} \mathbf{fv}(\texttt{p}) \setminus \{\texttt{id}\} \\
\mathbf{fv}(\texttt{(lim n p)}) &\stackrel{\text{def}}{=} \mathbf{fv}(\texttt{p})
\end{aligned}
$$

The auxiliary $\mathbf{fvv}(\mathtt{v})$ function returns the set of free variables in a given value v:

$$\mathbf{fvv}(\mathtt{[]}) \stackrel{\text{def}}{=} \varnothing$$
$$\mathbf{fvv}(\mathtt{id}) \stackrel{\text{def}}{=} \{\mathtt{id}\}$$

7.3 Abstract machine

David N. Turner in his PhD thesis [20] describes implementable abstract machine for a useful subset of the pi-calculus. It is a final link in a sequence of abstract machines. With each step his abstract machine is closer to an efficient implementation.

We try to achieve our goal (enabling memory accountability) by taking Turner's abstract machine and making yet another refinement. The modified abstract machine is described in this section. Our refinement makes the whole abstract machine more complex but in an important aspect it is also more useful.

Our abstract machine has higher number of reduction rules than Turner's because:

- We had to define semantics of the new (lim n p) construct.
- In addition to existing channel states we introduce a new channel state called *sink*. It is denoted as $\otimes$. This adds one whole line to Table 1.
- In addition to existing communication primitives we introduce a new one that is called *donor output*. It is denoted as $. This adds one whole column to Table 1.
- In our version of reduction rules we designate the owner of all newly created objects with respect to owners of already existing objects that interact in a given reduction rule.

Let *Domains* be a set of all domains of trust that can be created with the lim construct. Let *Identifiers* be all variable identifiers that can occur in programs. Let *Values* be the set of all values to which variables can be bound. It contains a *unit value* (denoted as []) and *Capabilities* to channels. Let *Channels* denote a set of channels that can be constructed as described in Section 7.5.

7.4 Domain variables

Reduction rules of our abstract machine define:

- what action should be performed in various different situations,
- how will system's state change when we perform that action,
- to which domain will belong newly created objects.

In this section we describe how do we denote owners of objects.

In our reduction rules, we bind *domain variables* to domains of existing objects that take part in the reduction. We use those variables to assign ownership of newly created objects. Let us explain the notation. In the following expression:

$$\overline{\text{object}_1}^{\alpha}$$

domain variable α is bound to the domain to which object_1 belongs. Similarly, in the following expression:

$$\overline{\text{object}_2}^{\omega}$$

domain variable ω is bound to the domain to which object_2 belongs. Different domain variables can be (by coincidence) bound to the same domain. In general, different domain variables are bound to different domains.

There are many situations in which objects belong to some domain but we do not want to bind any new domain variable because the domain of that object is irrelevant. In this case we decorate this object with a simple line:

$$\overline{\text{object}_3}$$

We do not omit the line completely because it consistently reminds us that object_3 also belongs to some domain.

Domain variables are always bound to values taken from the *Domains* set.

7.5 Asynchronous channel queues

Asynchronous channel may be in one of the following five states:

- An empty channel: $\bullet$
- A sequence of writers: $!v_1 :: \ldots :: !v_m$
  Individual values v_i were sent by some processes to this channel and can be read from it.
- A sequence of non-replicated readers:

  $$(E_1, ?\mathtt{x_1}{=}\mathtt{p_1}) :: \ldots :: (E_n, ?\mathtt{x_n}{=}\mathtt{p_n})$$

  Single reader is denoted as $(E, ?\mathtt{x}{=}\mathtt{p})$ where E is an environment capturing binding of all free variables of p, and p is a process term that is executed if this reader indeed receives some value. Received value is bound to variable x. Non-replicated reader is **always removed** from the channel if it receives a value.
- A single replicated reader: $(E, ?{*}\mathtt{x}{=}\mathtt{p})$.
  Here E is an environment capturing binding of its free variables of p, and p is a process term that is executed if this reader indeed receives some value. Received value is bound to variable x. Replicated reader is **never removed** from the channel if it receives a value.
- a sink
  (denoted as $\otimes$)

Existing channels may be turned into sink by the *kill* operation. This special channel state does not exist in the original Pict programming language implementation. Any value sent to a sink channel is thrown away instead of queuing in the channel. All readers that try to read from a sink channel can be thrown away because no value can be ever received from a sink. Channels of particular type are here denoted in the following way:

$$
\begin{array}{rcl}
\overline{C}^{\alpha} & ::= & \overline{\bullet}^{\alpha} \\
 & | & \overline{\overline{!v_1} :: \ldots :: \overline{!v_m}}^{\alpha} \\
 & | & \overline{\overline{(E_1, ?\mathtt{x_1}=\mathtt{p_1})} :: \ldots :: \overline{(E_n, ?\mathtt{x_n}=\mathtt{p_n})}}^{\alpha} \\
 & | & \overline{\overline{(E, ?{*}\mathtt{x}=\mathtt{p})}}^{\alpha} \\
 & | & \otimes
\end{array}
$$

All elements of the *Channel* set can be constructed this way. We track owner of the whole channel as well as owners of particular readers or writers queued in the channel.

7.6 Environments

Environments map variable names to the corresponding values. We can model them as functions with the following type:

$$\textit{Identifiers} \rightarrow \textit{Values}$$

We can write environments literally as $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$.

7.7 Machine state

Machine state is a triple (Q, H, R). Here Q maps particular domains to their corresponding memory quotas. We can model it as a function with the following type:

$$\textit{Domains} \rightarrow \mathbb{N}$$

H represents so called *heap*. It maps capabilities to channels. We can model it by an injective function with the following type:

$$\textit{Capabilities} \rightarrow \textit{Channels}$$

R is so called *run queue*. It is a sequence of thread contexts $\overline{(E_i, \mathtt{p}_i)}$ where E_i is an environment mapping free variables of the process term p to corresponding values. We write run queue as follows:

$$
\begin{array}{rcl}
R & ::= & \bullet \\
 & | & \overline{(E_1, \mathtt{p_1})} :: \overline{(E_2, \mathtt{p_2})} :: \ldots :: \overline{(E_n, \mathtt{p_n})}
\end{array}
$$

Empty run queue is denoted as $\bullet$.

7.8 Operations with functions

Several important components of the abstract machine are represented as functions. Different operations (function application $f(v)$, function override $f \oplus g$, domain restriction $S \lhd f$ and range restriction $f \rhd S$) are written in the Z notation [23].

7.9 Heap usage

Our compiler [7] converts all process terms into a set of *threads*. By thread we mean a finite sequence of primitive actions from a core language. How are particular process terms broken into threads is defined by the reduction rules. These threads can then at runtime invoke each other in a way we cannot predict at compile time. However, since at compile time we know exactly from what primitive actions are particular threads composed, for every thread we can compute the number of words it allocates from the heap of free memory. We can define *heap_usage* function that computes this number for a thread that corresponds to any given process term p. The definition of this function is dependent on internal representation of objects created during execution of threads.

7.10 Reduction and thread

The following judgment

$$Q, H, R \rightarrow Q', H', R'$$

states that machine in state Q, H, R evolves into machine in state Q', H', R'. The transition happens after finite number of actions (so called *thread*) defined by relation $\Downarrow$.

Relation

$$Q, H, E, \overline{\mathtt{p}}, R \Downarrow Q', H', R'$$

formally defines what is a *thread*. For every machine state Q, H, R and every thread context $\overline{(E, \mathtt{p})}$ it defines new state Q', H', R' after we execute the whole thread (i.e. all the actions that can be performed until p either terminates or blocks).

7.11 Reduction rules

To illustrate completeness of reduction rules, we can divide them into two groups:

- rules in the first group are not directly related to message-passing: SCHED$_1$, SCHED$_2$, SCHED$_3$, LIM$_1$, LIM$_2$, NIL, PRL and NEW. These determine the scheduling policy, the semantics of `lim` construct, null process, parallel composition of processes and what it means if we try to create a new channel.
- rules in the second group are related to message-passing: AOUT-R, AOUT-W, AOUT-R$*$, AOUT-S, ADOUT-R, ADOUT-W, ADOUT-R$*$, ADOUT-S, INP-R, INP-W, INP-R$*$, INP-S, REPL-E

Completeness of the first group is obvious. Completeness of the second group can be checked with Table 1. Which particular rule is applicable depends on:

- the kind of operation we try to perform: `!` (output), `$` (donor-output), `?` (input), `?*` (replicated-input); there is one column for each action

channel state	currently intended action: x!y	x$y	x?y=p	x?*y=p
empty	AOut-W	ADOut-W	Inp-R	Repl-E
writers	AOut-W	ADOut-W	Inp-W	$\times$
readers	AOut-R	ADOut-R	Inp-R	$\times$
repl. reader	AOut-R$*$	ADOut-R$*$	Inp-R$*$	$\times$
sink	AOut-S	ADOut-S	Inp-S	$\times$

Table 1. Reduction rules related to reading from or writing to channels in various states. Situations marked as $\times$ are excluded.

- state of the channel: empty, containing some writers, containing some non-replicated readers, containing a single replicated reader or sink; there is one row for each channel state.

Scheduler: The $\textsc{Sched}_1$, $\textsc{Sched}_2$, and $\textsc{Sched}_3$ rules define scheduling policy. Current scheduling policy is simple. We always pick the first runnable thread from the *run queue*.

- If there is enough free space to run the chosen thread, we simply run it ($\textsc{Sched}_1$).
- If there is not enough free space, we run the garbage collector. If then there is enough space to run the chosen thread, we run it ($\textsc{Sched}_2$).
- If, after running the garbage collector, there is not enough space, then there is no doubt that some of the domains allocated (and retained) more memory than its memory quota. We invoke the kill operation. It frees enough space. Then we run the chosen thread ($\textsc{Sched}_3$).

The *heap_usage* function was described in Section 7.9. The *free_space* function returns the size of the remaining free space. In our implementation [7] it has O(1) time complexity.

$$\frac{\begin{array}{l} \mathit{heap_usage}(\texttt{p}) \le \mathit{free_space}(H,R) \\ \qquad Q,H,E,\overline{\texttt{p}}^{\alpha},R \Downarrow Q',H',R' \end{array}}{Q,H,\overline{(E,\texttt{p})}^{\alpha} :: R \longrightarrow Q',H',R'}\ \textsc{Sched}_1$$

$$\frac{\begin{array}{l} \mathit{free_space}(H,R) < \mathit{heap_usage}(\texttt{p}) \\ \quad H' = \mathit{garbage_collect}(H,\overline{(E,\texttt{p})}^{\alpha} :: R) \\ \qquad \mathit{heap_usage}(\texttt{p}) \le \mathit{free_space}(H',R) \\ \qquad\quad Q,H',\overline{(E,\texttt{p})}^{\alpha} :: R \Downarrow Q'',H'',R'' \end{array}}{Q,H,\overline{(E,\texttt{p})}^{\alpha} :: R \longrightarrow Q'',H'',R''}\ \textsc{Sched}_2$$

$$\frac{\begin{array}{l} \mathit{free_space}(H,R) < \mathit{heap_usage}(\texttt{p}) \\ \quad H' = \mathit{garbage_collect}(H,\overline{(E,\texttt{p})}^{\alpha} :: R) \\ \quad \mathit{free_space}(H',R) < \mathit{heap_usage}(\texttt{p}) \\ \qquad (Q'',H'',\overline{(E,\texttt{p}'')}^{\alpha} :: R'') = \mathit{kill}(Q,H',\overline{(E,\texttt{p})}^{\alpha} :: R) \\ \qquad\quad Q'',H'',\overline{(E,\texttt{p}'')}^{\alpha} :: R'' \Downarrow Q''',H''',R''' \end{array}}{Q,H,\overline{(E,\texttt{p})}^{\alpha} :: R \longrightarrow Q''',H''',R'''}\ \textsc{Sched}_3$$

Note that we do not have to check whether

$$\mathit{heap_usage}(\texttt{p}'') \le \mathit{free_space}(H'',R'')$$

holds immediately after *kill* is completed. The *kill* operation itself ensures that this condition will be met.

Execution of threads: $\textsc{Lim}_1$ is the first of two rules that defines behavior of the $\overline{\texttt{(lim p n)}}^{\alpha}$ process. It handles a case when the original domain α tries to create a new domain with a larger quota then its own quota. In this case, the $\overline{\texttt{(lim p n)}}^{\alpha}$ process is simply ignored—it behaves as a null process.

$$\frac{q = Q(\alpha) - \texttt{n} \qquad q < 0}{Q,H,E,\overline{\texttt{(lim p n)}}^{\alpha},R \Downarrow Q,H,R}\ \textsc{Lim}_1$$

If some domain α rule tries to create a domain with a smaller memory quota then its own, this operation succeeds. The $\textsc{Lim}_2$ defines behavior of the abstract machine for this case. New domain is created, memory quotas Q are updated and process p is run in the new domain.

$$\frac{\begin{array}{l} q = Q(\alpha) - \texttt{n} \\ \quad 0 \le q \\ \qquad \omega \text{ is bound to a fresh domain} \\ \qquad\quad Q \oplus \{\alpha \mapsto q, \omega \mapsto \texttt{n}\},H,E,\overline{\texttt{p}}^{\omega},R \Downarrow Q',H',R' \end{array}}{Q,H,E,\overline{\texttt{(lim p n)}}^{\alpha},R \Downarrow Q',H',R'}\ \textsc{Lim}_2$$

$\textsc{Nil}$ rule is applied if we have to execute the null process. It is discarded and has no effect on the state of the machine.

$$\frac{}{Q,H,E,\overline{\texttt{()}},R \Downarrow Q,H,R}\ \textsc{Nil}$$

$\textsc{Prl}$ rule is applied if we have to execute a parallel composition `(p|q)` of two processes `p` and `q`. Current thread continues by executing first process `p` and then process `q`. This is stricter but still consistent with the meaning of a parallel composition of two pi-calculus processes. Both processes are executed in the same domain to which original process `(p|q)` belonged.

$$\frac{Q,H,E,\overline{\mathtt{p}}^{\alpha},R \Downarrow Q',H',R' \qquad Q',H',E,\overline{\mathtt{q}}^{\alpha},R' \Downarrow Q'',H'',R''}{Q,H,E,\overline{(\mathtt{p|q})}^{\alpha},R \Downarrow Q'',H'',R''}\ \textsc{Prl}$$

New rule is applied if some process wants to create a new channel. The new channel will belong to the same domain to which belongs the creator process. Current thread continues by execution of the process p in an enriched environment.

$$\frac{c \text{ is a fresh capability} \qquad Q,H\oplus\{c\mapsto\overline{\bullet}^{\alpha}\},E\oplus\{\mathtt{x}\mapsto c\},\overline{\mathtt{p}}^{\alpha},R \Downarrow Q',H',R'}{Q,H,E,\overline{(\mathtt{new\ x\ p})}^{\alpha},R \Downarrow Q',H',R'}\ \textsc{New}$$

Inp-S rule says that if some process tries to receive a value from the sink, that process is thrown away. That is plausible because no value, ever, can be received from the sink.

$$\frac{E(\mathtt{x})=c \qquad H(c)=\otimes}{Q,H,E,\overline{\mathtt{x?y=p}},R \Downarrow Q,H,R}\ \textsc{Inp-S}$$

Inp-R$*$ rule says that if some process tries to receive a value from a channel that already contains a replicated reader, that process will be thrown away. Replicated reader simply receives all the messages sent to this channel.

$$\frac{E(\mathtt{x})=c \qquad H(c)=\overline{\overline{(E,\mathtt{?*z=p})}}}{Q,H,E,\overline{\mathtt{x?y=p}},R \Downarrow Q,H,R}\ \textsc{Inp-R}*$$

Inp-W rule is applied if some process x?y=p is trying to receive a value from a channel that already contains some writer $\mathtt{!}v$. In this case we have two reasonable options for assigning ownership of the continuation p:

1. it may belong to the same domain as process x?y=p
2. or it may belong to the same domain as writer $\mathtt{!}v$

The second option would make servers vulnerable to clients. Clients invoke servers by sending messages to certain channels. Servers read these messages, perform appropriate actions and usually also reply back to clients. This rule determines which domain will own processes that are activated when server handles a request. If we choose the second option, then activated processes that handle client's request would be owned by client's domain. This is not desirable because if the client dies for some reason, the processes which work on its behalf in the server will be killed. This may break invariants in the server and may leave the server in a state when it cannot serve other clients. We have therefore chosen the first option. The death of the client will not disrupt completion of its former request. The result of the request is computed and sent to the (sink) channel. The server is afterward ready to serve other requests.

$$\frac{E(\mathtt{x})=c \qquad H(c)=\overline{\overline{\mathtt{!}v}::ws} \qquad Q,H\oplus\{c\mapsto ws\},E\oplus\{\mathtt{y}\mapsto v\},\overline{\mathtt{p}}^{\alpha},R \Downarrow Q',H',R'}{Q,H,E,\overline{\mathtt{x?y=p}}^{\alpha},R \Downarrow Q',H',R'}\ \textsc{Inp-W}$$

Inp-R rule is applied if some process x?y=p tries to read a value from a channel that already contains some blocked non-replicated readers. It will block too and its owner domain will not change. That means that the domain that added a new non-replicated reader to the channel will be billed for the space that that reader occupies—regardless of who owns the channel. The F environment contains only those bindings from E that are relevant for process p (values of its free variables). The $\mathbf{fv}(\mathtt{p})$ function is defined in Section 7.2.

$$\frac{E(\mathtt{x})=c \quad H(c)=\overline{rs}^{\omega} \quad F=(\mathbf{fv}(\mathtt{p})\setminus\{\mathtt{y}\})\lhd E}{Q,H,E,\overline{\mathtt{x?y=p}}^{\alpha},R \Downarrow Q,H\oplus\{c\mapsto \overline{rs::\overline{(F,\mathtt{?y=p})}^{\alpha}}^{\omega}\},R}\ \textsc{Inp-R}$$

AOut-S and ADOut-S rules are applied if some process tries to send a value to a sink channel. In both cases the value written to the sink channel is discarded.

$$\frac{E(\mathtt{x})=c \qquad H(c)=\otimes}{Q,H,E,\overline{\mathtt{x!z}},R \Downarrow Q,H,R}\ \textsc{AOut-S}$$

$$\frac{E(\mathtt{x})=c \qquad H(c)=\otimes}{Q,H,E,\overline{\mathtt{x\$z}},R \Downarrow Q,H,R}\ \textsc{ADOut-S}$$

If some writer x$z or x!z is trying to write a value to a channel that contains a non-replicated reader $(F,\mathtt{?y=p})$ then the reader is resumed and it will be moved from the channel at the end of the run queue. The environment of the reader is enriched with the received value. Concerning the owner of the resumed process, two reasonable policies can be considered:

1. resumed process should be owned by reader's domain
2. resumed process should be owned by writer's domain

If we consider client-server scenario described in connection with reduction rule Inp-R, we must choose the first option.

$$\frac{E(\mathtt{x}) = c \qquad H(c) = \overline{\overline{(F, \mathtt{?y=p})}^{\alpha} :: rs}^{\omega}}{Q, H, E, \overline{\mathtt{x!z}}, R \Downarrow Q, H \oplus \{c \mapsto \overline{rs}^{\omega}\}, R :: \overline{(F \oplus \{\mathtt{y} \mapsto E(\mathtt{z})\}, \mathtt{p})}^{\alpha}} \;\textsc{AOut-R}$$

$$\frac{E(\mathtt{x}) = c \qquad H(c) = \overline{\overline{(F, \mathtt{?y=p})}^{\alpha} :: rs}^{\omega}}{Q, H, E, \overline{\mathtt{x\$z}}, R \Downarrow Q, H \oplus \{c \mapsto \overline{rs}^{\omega}\}, R :: \overline{(F \oplus \{\mathtt{y} \mapsto E(\mathtt{z})\}, \mathtt{p})}^{\alpha}} \;\textsc{ADOut-R}$$

If some writer `x!z` tries to write a value to a channel that already contains some writers, it is appended at the end of the channel queue. Concerning the owner of the appended writer, there are two plausible policies:

1. writer should be owned by the same domain that sent a given value to a the channel
2. writer should be owned by the same domain that owns the channel

Both policies are plausible in different situations. The programmer can choose one or the other policy (if type system permits) explicitly by performing either ordinary output action `x!z` or donor output action `x$z`. Each is handled by one of the following two separate reduction rules:

$$\frac{E(\mathtt{x}) = c \qquad H(c) = \overline{ws}^{\omega}}{Q, H, E, \overline{\mathtt{x!z}}^{\alpha}, R \Downarrow Q, H \oplus \{c \mapsto \overline{ws :: \overline{\mathtt{!}E(\mathtt{z})}^{\alpha}}^{\omega}\}, R} \;\textsc{AOut-W}$$

$$\frac{E(\mathtt{x}) = c \qquad H(c) = \overline{ws}^{\omega}}{Q, H, E, \overline{\mathtt{x\$z}}, R \Downarrow Q, H \oplus \{c \mapsto \overline{ws :: \overline{\mathtt{!}E(\mathtt{z})}^{\omega}}^{\omega}\}, R} \;\textsc{ADOut-W}$$

Repl-E rule applies if some replicated reader tries to read from (i.e. listen to) some channel. The language properties enable us to omit certain checks. We in advance know that the channel must be owned by the same domain to which replicated reader belongs and we in advance know that the channel is empty.

$$\frac{E(\mathtt{x}) = c \qquad F = \mathbf{fv}(\mathtt{p}) \lhd E}{Q, H, E, \overline{\mathtt{x?*y=p}}^{\alpha}, R \Downarrow Q, H \oplus \{c \mapsto \overline{\overline{(F, \mathtt{?*y=p})}^{\alpha}}^{\alpha}\}, R} \;\textsc{Repl-E}$$

If some writer tries to write a value to a channel that contains a replicated reader then the resumed reader is copied[10] at the end of the run queue. It will be executed in an enriched environment that captures also the received value. The original replicated reader **is never removed** from the channel. Concerning the owner of the resumed process, there are two plausible policies:

1. resumed process should be owned by replicated reader's domain
2. resumed process should be owned by writer's domain

The client-server scenario described with the Inp-W justifies our decision to choose the first option.

$$\frac{E(\mathtt{x}) = c \qquad H(c) = \overline{\overline{(F, \mathtt{?*y=p})}^{\alpha}}^{\alpha}}{Q, H, E, \overline{\mathtt{x!z}}, R \Downarrow Q, H, R :: \overline{(F \oplus \{\mathtt{y} \mapsto E(\mathtt{z})\}, \mathtt{p})}^{\alpha}} \;\textsc{AOut-R}*$$

$$\frac{E(\mathtt{x}) = c \qquad H(c) = \overline{\overline{(F, \mathtt{?*y=p})}^{\alpha}}^{\alpha}}{Q, H, E, \overline{\mathtt{x\$z}}, R \Downarrow Q, H, R :: \overline{(F \oplus \{\mathtt{y} \mapsto E(\mathtt{z})\}, \mathtt{p})}^{\alpha}} \;\textsc{ADOut-R}*$$

None of the above reduction rules defines the meaning of primitive processes `(ccode ...)`. For each primitive process we need a separate ad-hoc axiom that defines its meaning.

Neither is any of reduction rules applicable if run queue is empty. In this case no other reductions are possible and abstract machine halts regardless of the contents of the heap H.

8. Related work

PLT Scheme project [22] dealt with a similar problem we are trying to address. However, they decided to enforce *consumer-based memory accounting*. We support both schemes: the *producer-based memory accounting* by our *output action* and *consumer-based memory accounting* with our *donor-output action*.

Various groups focus on functional programs, c.f. [21]. Pi-calculus is a more complicated model of describing behavior then lambda-calculus but it is also more expressive one. We have decided not to trade this expressibility for other properties which are not essential: the ability to determine the worst case heap space usage at compile time. If it is sufficient to recognize excess memory usage at runtime, we are not forced to rely on a language with restricted expressive power.

It was already shown [16] how to achieve defensive correctness among concurrent processes interacting via *synchronous rendezvous*. The proposed updates of the Pict

[10] Turner [20] also shows how to avoid this copying. He introduces notion of *local environment* and *global environment*.

programming language enable us to achieve defensive correctness among concurrent processes interacting via *asynchronous message-passing over channels*.

Channel contracts in Singularity [17] addresses a different set of problems then we are trying to address. Our primary goal is to avoid DoS attacks. The original purpose of contracts in Singularity is to ensure freedom from deadlock.

9. Conclusion and future work

We have partially validated usefulness of the proposed language constructs by showing how assumed attacks can be countered [7]. Complete validation can be done only with a rich language with usual set of basic types, type constructors, syntactic sugar and extralinguistic features. If they are reconcilable with the proposed core language, then the rich language can be used for creating a defensively correct operating system with a smallest trusted computing base.

References

[1] Google Caja.
`http://code.google.com/p/google-caja`.

[2] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Countering IPC Threats in Multiserver Operating Systems. In *Accepted for publication at 14th Pacific Rim International Symposium on Dependable Computing (PRDC'08)*, Taipei, Taiwan, Dec. 2008.

[3] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1243418.1243424.

[4] G. C. Hunt, J. R. Larus, D. Tarditi, and T. Wobber. Broad new os research: challenges and opportunities. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.

[5] M. Košík. Taming of Pict. In V. Geffert, J. Karhumäki, A. Bertoni, B. Preneel, P. Návrat, and M. Bieliková, editors, *SOFSEM*, volume 4910 of *Lecture Notes in Computer Science*, pages 610–621. Springer, 2008. ISBN 978-3-540-77565-2.

[6] M. Košík. Backwater kernel home page, 2009.
`http://altair.sk/mediawiki/index.php/Backwater`.

[7] M. Košík. Kalahari: Experimental Programming Language Based on Pict, 2009.
`http://altair.sk/mediawiki/index.php/Kalahari`.

[8] M. Košík. Sandboxed Ping Program. 2009. URL `http://altair.sk/mediawiki/index.php/SandboxedPing`.

[9] A. M. Mettler and D. Wagner. The Joe-E Language Specification (draft). Technical Report UCB/EECS-2006-26, EECS Department, University of California, Berkeley, 3 2006.

[10] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[11] B. C. Pierce. Programming in the pi-calculus: A tutorial introduction to Pict. 1997. URL `http://citeseer.ist.psu.edu/pierce97programming.html`.

[12] B. C. Pierce and D. N. Turner. Pict libraries manual. 1997.

[13] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000. ISBN ISBN 0-262-16188-5.
`http://citeseer.ist.psu.edu/pierce97pict.html`.

[14] B. C. Pierce and D. N. Turner. Pict language definition. 1997. URL `http://citeseer.ist.psu.edu/article/pierce96pict.html`.

[15] D. Sangiorgi and D. Walker. *The Pi-calculus: A Theory of Mobile Processes*. Cambridge University Press, July 2001. ISBN 0521781779.

[16] J. S. Shapiro. Vulnerabilities in synchronous IPC designs. In *In Proc. IEEE Symposium on Security and Privacy*, pages 251–262. IEEE Computer Society Press, 2003.

[17] Z. Stengel and T. Bultan. Analyzing singularity channel contracts. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 13–24, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-338-9. doi: http://doi.acm.org/10.1145/1572272.1572275.

[18] M. Stiegler. Emily: A high performance language for enabling secure cooperation. In *C5 '07: Proceedings of the Fifth International Conference on Creating, Connecting and Collaborating through Computing*, pages 163–169, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2806-6. doi: http://dx.doi.org/10.1109/C5.2007.13.

[19] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation*. Pearson Prentice Hall, 2006.

[20] D. N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.

[21] P. Unnikrishnan, G. Chen, M. Kandemir, M. Karakoy, and I. Kolcu. Reducing memory requirements of resource-constrained applications. *ACM Trans. Embed. Comput. Syst.*, 8(3):1–37, 2009. ISSN 1539-9087. doi: http://doi.acm.org/10.1145/1509288.1509289.

[22] A. Wick and M. Flatt. Memory accounting without partitions. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 120–130, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4. doi: http://doi.acm.org/10.1145/1029873.1029888.

[23] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-948472-8.
`http://www.usingz.com`.