

Typed Static Analysis for Concurrent, Policy-Based, Resource Access Control

Nicholas Nguyen, Julian Rathke

Department of Informatics
University of Sussex
{nn20, julianr}@sussex.ac.uk

Abstract. We present a type and effect system for statically determining whether concurrent programs in a simple functional language adhere to a strict access control policy. Policy states are represented by automata states and are tracked, statically, by the type and effect system. We ensure that, per thread, all function calls are, independently, in accordance with policy with respect to the current state. To verify that several concurrent threads jointly satisfy a policy we are required to explore the state space of an interleaving of the several threads' effects. However, we observe that by furnishing our language with monitor synchronisation primitives we can reduce our state space search by abstracting away behaviours inside atomic sections. The type and effect system is proved to satisfy Subject Reduction and a Safety theorem which guarantees that well-typed programs never violate policy.

1 Introduction

Resource access control has its origins in the field of operating systems research [14] but more recently has gained attention for use in programming language research [23, 16, 10, 6, 8]. With the recent growth of the global computing paradigms, this is particularly true of programming languages to support distributed applications. The essence of resource access control is to enable untrusted programs to execute in a system in a manner prescribed by the system policy. Systems policies represents allowable behaviours by third party code and can range in their sophistication from simple access matrices [14] to sophisticated RBAC schemes etc [9, 7]. Policies may not only protect access to privileged resources but may also ensure that code respects dynamic behaviours in the form of specified protocols. In this paper we do not focus on the policy specification but rather on the policy enforcement mechanism.

A typical approach to policy enforcement is to instrument the operating system or virtual machine in which the controlled code is running to monitor resource access at runtime. This approach can generate undesirable overheads and, for this reason, alternative approaches using static verification techniques such as strong typing have also been investigated e.g. [22, 13, 11, 17, 1, 5, 21]. We consider this latter approach for systems of concurrent threads.

Static analysis for resource access control for sequential code is becoming reasonably well understood [3, 8]. For concurrent systems we can use capability (sub)typing for an analysis of simple forms of access permissions [19, 11]. However, to study policy based access control more thoroughly we need to consider more general policy specifications addressing dynamic behaviours as in for example [13]. The difficulties in verifying adherence to dynamic policies in a concurrent system arise due to the fact that a policy on shared resources accessed by concurrent threads must be *jointly* satisfied. It is not sufficient to verify that each thread obeys the policy independently. We must consider all possible interleavings of resource accesses by the separate threads in order to capture the runtime behaviour of programs more accurately.

In this paper we make use of type and effect systems [15, 1, 21] to extract effects as models, and to statically check that each thread separately satisfies a global policy before using a reduced state space exploration of the interleaved behaviour models. The particular language we use is a simply-typed λ -calculus augmented with constructs for thread creation and Java style monitor primitives with synchronized expressions. We make use of these by noticing that within a monitor the thread has mutually exclusive access to system resources and, by virtue of this, it is sufficient to verify the policy state only at strategic points within transactions, such as monitor entry points. This observation helps to reduce the size of the state space when it comes to checking interleavings of threads.

We decorate behaviour models during model extraction with summaries which instruct the model checker on how to track the policy state. A summary takes the form of a mapping, Ψ , which relates the policy state at the current point of execution, backwards, to the state at the beginning of either the current transaction, or, the most recent function call.

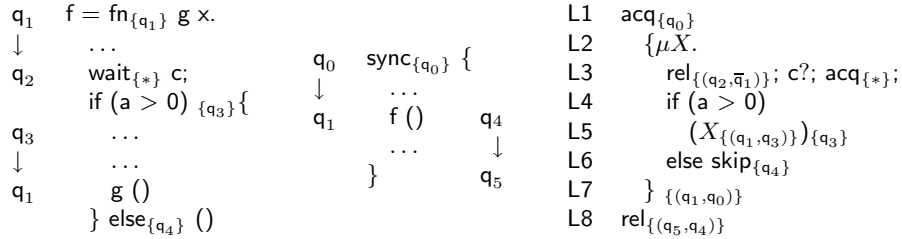


Fig. 1. A policy(left), a client function(centre) and its latent behaviour model(right).

The automata we use to represent basic forms of policy lack the sophistication of some policy specification languages [9, 7]. However, we do not study policy specification, but rather, static enforcement of policy on systems of concurrent threads.

An outline of our main technical contribution in Figure 1 includes the key features of our source language and the behaviour models that we extract from

source programs. The behaviour model in the right of Figure 1 is extracted from a synchronized block of code (centre) which makes a call to the function f (left). The annotation at the start of the synchronized block asserts that the policy state must be q_0 when the monitor is entered. The type system uses this annotation to verify the resources accesses (represented by the dotted lines) from state q_0 , up to state q_1 at the point of the call to f . The model checker must later validate this assumption whenever the behaviour model acquires the monitor lock at line L1. The transaction beginning at state q_0 ends inside f at state q_2 . The wait command releases the monitor lock, blocks until $c > 0$, and may re-enter the monitor in any state, represented by $\{*\}$. The summary from state q_1 to q_2 is annotated onto the release action of the model at line L3, by the pair (q_2, \bar{q}_1) . The marked state \bar{q}_1 symbolizes that the current transaction ends inside f and that a summary from both of f 's calling contexts is required firstly at line L7 and then at line L5.

After re-entering the monitor, the assertion check $\{*\}$ is postponed until after the conditional check on variable a , which associates state q_3 with the predicate $a > 0$ and q_4 with the predicate $a \leq 0$. These refined assertions are checked in the model at lines L5 and L6. Function f returns in state q_4 and the final summary maps q_5 to q_4 at monitor exit on line L8.

In summary, our type and effect system performs an interprocedural analysis which tracks the state of the policy across the boundaries of transactions and function calls. The resource accesses are abstracted from the behaviour models and replaced with summaries of policy state changes.

1.1 Related work

A clear precedent for this work is the type and effect system of [1]. We extend the system presented there for generating summaries of the policy state changes off the effects. An approach to static verification which is proving of great interest is that of software model checking. Solid advances in software verification have already been made for sequential languages in the SLAM project [3]. SLAM extracts boolean programs from C code yielding highly accurate runtime behaviours. Resource accesses within programs often depend on control decisions based on the values of program variables. Representing these values in the abstract models is often key to yielding more true positives through the analyses. We hypothesize that it is fruitful to represent program state more directly within program models, such as with the integer variables we use here. For example k signals can be used to represent the fact that resource π can be accessed k times in succession, whereas it is not clear how to represent this fact with boolean programs.

Procedure summarization [20] within the Zing software model checker [2] is closely related to the work reported here. In their work the effects of procedures on program variables are summarized across atomic sections and call stack boundaries. Their approach and ours, both encounter interesting interactions between the boundaries of atomic sections and procedure/function bodies. The main difference between our approaches however, is that Zing generates

summaries at the time of model checking, whereas in our setting the finite state of the policy allows summaries to be statically generated during model extraction.

2 Syntax and Reduction Semantics

Automata provide a basic specification model for resource access control where the automaton's alphabet represents resources and the transitions represent the rules for accessing resources. We model resources as named function calls in a λ -calculus. We consider a resource π to have been accessed when a function explicitly labelled with the name π is applied to some argument.

Definition 1. *A policy automaton \mathcal{P} is a triple $(\text{States}, \text{Funclds}, \delta)$ where States is a finite set of states, containing a special state `dead`, and Funclds is a set of labels which contains a special name ϵ . The transition function $\delta : \text{States} \times \text{Funclds} \rightarrow \text{States}$ is such that $\delta(q, \epsilon) = q$ for all $q \in \text{States}$.*

We use ϵ to label functions which are not considered to be resources of the system. We use the `dead` state to model unavailability of a resource. To enforce a policy we need to prevent a program's sequence of function calls from driving the automaton into the `dead` state. Enforcement can be readily achieved through runtime monitoring of function calls by disallowing unsafe accesses at execution time. However, this incurs runtime overhead. In this paper we investigate the use of static analysis techniques to alleviate this overhead in a concurrent, functional language. The values of our language are boolean and unit values and functions.

Values	Expressions	Threads
	$e ::= e_1 e_2$	
	<code>inc(c)</code>	$T ::= T \parallel T \mid t e$
	<code>dec(c)</code>	$S \subseteq \text{States}$
$v ::= ()$	<code>if (c > 0) e_1 else e_2</code>	$t \in \text{Threadlds}$
<code>true</code>	<code>sync_S e</code>	$\pi \in \text{Funclds}$
<code>false</code>	<code>wait_S c</code>	$x, y, z, f, g \in \text{Vars}$
<code>fn_{πS} g x.e</code>	<code>_Se</code>	$c \in \text{NatVars}$
	<code>spawn e</code>	
	<code>v</code>	

We use an obvious syntax for recursive function. Functions are decorated with function identifiers π which also label the edges of the policy automata. Functions are also decorated with a set of states S , which declares the availability of this function as a resource. This is information for the static analysis and has no bearing on the runtime behaviour. The intended meaning of this is that the current policy state should be in S whenever the function π is called. Calling a function labelled with π and S when the policy state is q always drives the

policy into state $\delta(\mathbf{q}, \pi)$. All function calls made outside of the monitor must be labelled with ϵ .

Expressions include primitives for monitor synchronization with commands to wait on integer variables. The commands `inc` and `dec` increment and decrement the number of signals held on integer variables. The expression language is sufficient to encode arithmetic expressions and comparison operators on expressions. For example we can encode the expression $y = x - 1; \text{if } (x > y) e_1 \text{ else } e_2$. In order to keep the language simple, integer variables in `NatVars` are neither lexically bound nor dynamically created. Allowing references to integer variables would require our static analysis to perform an alias analysis, which is an important but orthogonal problem to the focus of this paper. Expressions protected by the monitor are those which occur under the scope of `syncS`. This expression is decorated with a set of policy states S , declaring that the expected policy state upon entering the critical section is in S . This is a hint for the static analyser only and does not affect the runtime behaviour. The remaining expressions include standard constructs for function application and thread spawning. A *system* T is built from the parallel composition of expressions e tagged with thread identifiers t .

2.1 Reduction Semantics

Evaluation of programs takes place with respect to a policy automata that exists as an entity in the runtime system, and whose state is tracked at runtime. Therefore, the reduction semantics is defined on tuples: $\langle \mathbf{q}; \star; \sigma; T \rangle$, where \mathbf{q} is the current state of the policy, \star represents the owner of the monitor lock and ranges over \emptyset and `ThreadIds`. When the lock is free then \star is \emptyset otherwise \star is t where t is the identifier of the thread which holds the lock. Signals on unsigned integer variables are held in $\sigma : \text{NatVars} \rightarrow \mathbb{N}$ where $\sigma(c)$ is the number of signals held for c . We let $(\sigma + c)(c') = \sigma(c) + 1$ if $c = c'$ otherwise $\sigma(c')$. T is a system of named parallel thread expressions.

The reduction rules specify a standard leftmost-outermost strategy for the λ -calculus [4], with a signal-and-continue synchronization monitor [12]. The rule `EV-ACQ` sends a thread t requesting the monitor into its critical section, symbolized by the runtime expression `in-sync e` and setting $\star = t$. When evaluation inside the critical section terminates with a value then t releases the lock as in the rule `EV-REL`. A thread t that needs to wait on c from inside its critical section, is made to exit the monitor by the rule `EV-W-REL`, and wait for a signal on c with the second form of runtime expression `waitS c ε[()]`. Thread t can continue to execute (rule `EV-WAIT`) by consuming a c signal from the buffer, and reverting to the expression `syncS .` The annotation S is carried in sequence between `in-sync ε[waitS c]`, `waitS c ε[()]` and `syncS ε[()]`. The remaining rules are standard.

The runtime errors of our language include misuse of the monitor primitives and violation of the resource access policy. Rule `ER1` indicates when a thread misuses the monitor by attempting to access an integer variable, or call a named function $\pi \neq \epsilon$, outside of the monitor. A policy violation error occurs by `ER2` when a function π is called from a state \mathbf{q} such that $\delta(\mathbf{q}, \pi) = \text{dead}$. A straightforward

Runtime expressions $e ::= \dots \mid \text{in-sync } e \mid \text{waiting}_S c \ e$
 Evaluation contexts $\varepsilon ::= [\bullet] \mid v \ \varepsilon \mid \varepsilon \ e \mid \text{in-sync } \varepsilon$
 $\mathcal{T}_t ::= T \parallel \mathcal{T}_t \mid \mathcal{T}_t \parallel T \mid t\varepsilon$

$\langle \mathbf{q}; \emptyset; \sigma; \mathcal{T}_t[\text{sync}_S e] \rangle$	$\rightarrow \langle \mathbf{q}; t; \sigma; \mathcal{T}_t[\text{in-sync } e] \rangle$	(EV-ACQ)
$\langle \mathbf{q}; t; \sigma; \mathcal{T}_t[\text{in-sync } v] \rangle$	$\rightarrow \langle \mathbf{q}; \emptyset; \sigma; \mathcal{T}_t[v] \rangle$	(EV-REL)
$\langle \mathbf{q}; t; \sigma; \mathcal{T}_t[\text{in-sync } \varepsilon[\text{wait}_S c]] \rangle$	$\rightarrow \langle \mathbf{q}; \emptyset; \sigma; \mathcal{T}_t[\text{waiting}_S c \ \varepsilon[()]] \rangle$	(EV-W-REL)
$\langle \mathbf{q}; \star; \sigma; \mathcal{T}_t[\text{waiting}_S c \ e] \rangle$	$\rightarrow \langle \mathbf{q}; \star; \sigma; \mathcal{T}_t[\text{sync}_S e] \rangle$ if $\sigma(c) > 0$	(EV-WAIT)
$\langle \mathbf{q}; \star; \sigma; \mathcal{T}_t[\text{spawn } e] \rangle$	$\rightarrow \langle \mathbf{q}; \star; \sigma; \mathcal{T}_t[()] \parallel t'e \rangle$ t' fresh	(EV-SPAWN)
$\langle \mathbf{q}; \star; \sigma; \mathcal{T}_t[se] \rangle$	$\rightarrow \langle \mathbf{q}; \star; \sigma; \mathcal{T}_t[e] \parallel t'e \rangle$	(EV-CHECK)
$\langle \mathbf{q}; \star; \sigma; \mathcal{T}_t[\text{inc}(c)] \rangle$	$\rightarrow \langle \mathbf{q}; \star; \sigma + c; \mathcal{T}_t[()] \rangle$	(EV-INC)
$\langle \mathbf{q}; \star; \sigma + c; \mathcal{T}_t[\text{dec}(c)] \rangle$	$\rightarrow \langle \mathbf{q}; \star; \sigma; \mathcal{T}_t[()] \rangle$	(EV-DEC)
$\langle \mathbf{q}; \star; \sigma; \mathcal{T}_t[\text{if } (c > 0) \ e_1 \ \text{else } e_2] \rangle$	$\rightarrow \langle \mathbf{q}; \star; \sigma; \mathcal{T}_t[e_k] \rangle$ $i = 1$ if $\sigma(c) > 0$ else $i = 2$	(EV-IF)
$\langle \mathbf{q}; \star; \sigma; \mathcal{T}_t[\text{fn}_{\pi S} \ \mathbf{g} \ \mathbf{x}.e \ v] \rangle$	$\rightarrow \langle \delta(\mathbf{q}, \pi); \star; \sigma; \mathcal{T}_t[e\{\text{fn}_{\pi S} \ \mathbf{g} \ \mathbf{x}.e/\mathbf{g}\}\{v/\mathbf{x}\}] \rangle$	(EV-FUN)
$\langle \mathbf{q}; \star; \sigma; \mathcal{T}_t[e] \rangle \searrow$	$\star \neq t$ and $e = \text{inc}(c), \text{dec}(c), \text{wait}_S c$	
	$v_1 \ v_2$ (v_1 is function $\pi \neq \epsilon$)	(ER ₁)
$\langle \mathbf{q}; \star; \sigma; \mathcal{T}_t[v_1 \ v_2] \rangle \searrow$	$v_1 = \text{fn}_{\pi S} \ \mathbf{g} \ \mathbf{x}.e$ and $\delta(\mathbf{q}, \pi) = \text{dead}$	(ER ₂)
$\langle \mathbf{q}; t'; \sigma; \mathcal{T}_t[\text{in-sync } e] \rangle \searrow$	$t' \neq t$	(ER ₃)

approach to ensure safe policy access uses the runtime check $\delta(\mathbf{q}, \pi) \neq \text{dead}$ on EV-FUN. This approach is less than satisfactory as it leads to an overhead of runtime checking. A more sophisticated approach is to use a static type system to track all function calls and check that these are in accordance with the policy. For the remainder of the paper we develop a sound static analysis based on type and effect systems of [1] for estimating when programs do not cause runtime errors, thus eliminating the need for runtime checks.

3 Effect behaviours

Our type and effect system extracts behavioural models from source programs in the sense of [3, 2]. Policy access errors are to be detected during model extraction and monitor misuse will be checked afterwards, on the manifest effects of expressions. The effect behaviour language Bhv is generated by \tilde{a} .

$a ::= \pi \mid c? \mid \text{inc}(c) \mid \text{dec}(c) \mid \text{acq} \mid \text{rel}$	$\text{MStates} = \{\bar{q} \mid q \in \text{States}\}$
$\tilde{a} ::= \tilde{a}; \tilde{a} \mid \text{spawn } \tilde{a} \mid \tilde{a}_S \mid \tilde{a}_\Psi$	$s \in \text{States} \cup \text{MStates}$
$\mid \text{if } (c > 0) \ \tilde{a}_1 \ \text{else } \tilde{a}_2$	$\text{States}^\triangleright = \text{States} \leftrightarrow \mathcal{P}(\text{States} \cup \text{MStates})$
$\mid \text{sync } \tilde{a} \mid \mu X. \tilde{a} \mid X \mid a$	$\Psi \in \text{States}^\triangleright$
$\tilde{t} ::= {}_t\tilde{a} \mid \tilde{t} \parallel \tilde{t}$	$X \in \text{RecVars}$

Our type and effect system checks sequential resource accesses with respect to the declarations of expected policy states at monitor entrances. These sequentially safe resource accesses, between a monitor entry and exit, are abstracted into a single effect rel_Ψ at release of the monitor's lock. We perform a state space exploration on concurrent behaviours in which policy accesses (accounted for in release effects) are further abstracted away, reducing the overall size of the models. The state space search checks the validity of the sequential checks with respect to concurrent behaviour, by checking that whenever a thread enters an atomic section then the policy state is contained in the declared set of expected states.

The effects acq and rel , acquire and release the monitor lock. The annotated effect \tilde{a}_S asserts that the tracked policy state is contained in S , just before evaluation of \tilde{a} begins. The annotated effect \tilde{a}_Ψ instructs the model checker to update the tracked policy state according to Ψ before evaluating \tilde{a} . Here, Ψ takes the form of a function from the possible current states to sets of states where $\Psi(\mathbf{q})$ represents the possible states the policy was in at the start of the current summary. Intuitively, $\mathbf{q}' \in \Psi(\mathbf{q})$ relates the current state \mathbf{q} , to state \mathbf{q}' at the beginning of the current transaction; and $\bar{\mathbf{q}}' \in \Psi(\mathbf{q})$ relates the current state \mathbf{q} at the end of the current transaction, to the state \mathbf{q}' at the beginning of the most recent function call. The set of states **States** and marked states **MStates** are disjoint. We write (\mathbf{q}, S) for mappings in Ψ when $\Psi(\mathbf{q}) = S$, and $(\mathbf{q}, \mathbf{q}')$ instead of $(\mathbf{q}, \{\mathbf{q}'\})$.

The effects $\text{inc}(c)$ and $\text{dec}(c)$ increment and decrement the integer value held for c . Sequential composition of behaviours reflects the sequential evaluation of expressions, and is right associative. The behaviour $\text{if } (c > 0) \tilde{a}_1 \text{ else } \tilde{a}_2$ is included from the source language. The behaviour $\text{sync } \tilde{a}$ delimits the scope of critical sections and is carried over into the syntax of behaviours to check proper monitor usage later on. The latent effect of functions is $\mu X.\tilde{a}$. We write $\text{unfold}(\mu X.\tilde{a})$ for the one step unfolding of latent effects. Thread behaviours \tilde{t} are the parallel composition of sequential behaviours tagged with thread identifiers.

3.1 Tracking state on behaviours

The type and effect system must keep track of the current policy state through all control flow paths of expressions. We therefore extend the automata transition function lifted to the language of effect behaviours Bhv . The lifted function δ has type:

$$\delta : (\text{States}^\triangleright \times Bhv \times (\text{RecVars} \rightarrow (\text{States}^\triangleright \rightarrow \text{States}^\triangleright))) \rightarrow \text{States}^\triangleright$$

and is defined by structural induction of behaviours.

For closed effects \tilde{a} and current state(s) Ψ , we have that $\delta(\Psi, \tilde{a}, \emptyset)$ returns Ψ' where Ψ' represents the new possible current states of the policy along with their mappings to sets of possible states at the last lock acquire if the program with effect \tilde{a} were to be executed inside a critical section. We write $\delta(\Psi, \tilde{a}) = \delta(\Psi, \tilde{a}, \emptyset)$.

$\delta(\Psi, \pi, F)$ updates the domain of Ψ according to the transition π from each $\mathbf{q} \in \text{dom}(\Psi)$. $\delta(\Psi, \text{acq}_S, F)$ jumps to the expected current states S , mapping each

$$\begin{array}{ll}
\delta(\Psi, \text{inc}(c), F) = \Psi & \delta(\Psi, \text{acq}_S, F) = \{(q, q) \mid q \in S\} \\
\delta(\Psi, \text{dec}(c), F) = \Psi & \delta(\Psi, \pi, F) = \bigcup \{(\delta(q, \pi), \Psi(q)) \mid q \in \text{dom}(\Psi)\} \\
\delta(\Psi, c?, F) = \Psi & \delta(\Psi, \text{if}(c > 0) \tilde{a}_1 \\
\delta(\Psi, \text{rel}, F) = \Psi & \quad \text{else } \tilde{a}_2, F) = \delta(\Psi, \tilde{a}_1, F) \cup \delta(\Psi, \tilde{a}_2, F) \\
\delta(\Psi, \text{spawn } \tilde{a}, F) = \Psi & \delta(\Psi, \tilde{a}_1; \tilde{a}_2, F) = \delta(\delta(\Psi, \tilde{a}_1, F), \tilde{a}_2, F) \\
\delta(\Psi, \text{sync } \tilde{a}, F) = \Psi & \delta(\Psi, X, F) = F(X)(\Psi) \\
\delta(\Psi, \tilde{a}_{\Psi'}, F) = \delta(\Psi, \tilde{a}, F) & \delta(\Psi, \mu X. \tilde{a}, F) = \text{Lfp}(\Theta)(\Psi) \text{ where} \\
\delta(\Psi, \tilde{a}_S, F) = \delta(S \cap \Psi, \tilde{a}, F) & \quad \Theta(g) = \lambda \Psi. \delta(\Psi, \tilde{a}, F \cup X \mapsto g)
\end{array}$$

$$\text{where } S \cap \Psi = \{(q', s) \mid s = q \text{ or } \bar{q} \ \& \ s \in \Psi(q') \ \& \ q \in S\}$$

state in S to itself. $\delta(\Psi, \tilde{a}_S, F)$ calculates a refinement of $\delta(\Psi, \tilde{a}, F)$ by intersecting the range of Ψ with S . Outside of critical sections, or for critical sections as a whole, this function provides little information and acts as the identity on `sync`. Since δ is monotone on the finite lattice $\text{States}^\triangleright$ with the pointwise ordering, then the least fix point of Θ , which defines $\delta(\Psi, \mu X. \tilde{a}, F)$, is known to exist.

3.2 Well-formed behaviours

The second form of runtime error occurs when either integer variables or resources protected by the policy are accessed outside of the monitor. It is easy to check an expression's monitor misuse statically, by stipulating that its behaviour's monitor actions and function calls, $\pi \neq \epsilon$, that change the policy's state, occur under the scope of `sync`. Well-formedness of behaviours also disallows the nesting of `sync`.

$$\begin{array}{c}
\text{WF}(\epsilon) \text{WF}(X) \frac{\text{WF}(\tilde{a})}{\text{WF}(\mu X. \tilde{a})} \frac{\text{WF}(\tilde{a}_1) \text{WF}(\tilde{a}_2)}{\text{WF}(\tilde{a}_1; \tilde{a}_2)} \frac{\text{WF}(\tilde{a})}{\text{WF}(\text{spawn } \tilde{a})} \frac{\text{Sync}(\tilde{a}) \text{FV}(\tilde{a}) = \emptyset}{\text{WF}(\text{sync } \tilde{a})} \\
\text{Sync}(X) \frac{\text{Sync}(\tilde{a})}{\text{Sync}(\mu X. \tilde{a})} \frac{\text{Sync}(\tilde{a})}{\text{Sync}(\tilde{a}_S)} \frac{\text{Sync}(\tilde{a})}{\text{Sync}(\tilde{a}_\Psi)} \\
\frac{\text{Sync}(\tilde{a}_1) \text{Sync}(\tilde{a}_2)}{\text{Sync}(\tilde{a}_1; \tilde{a}_2)} \frac{\text{Sync}(\tilde{a}_1) \text{Sync}(\tilde{a}_2)}{\text{Sync}(\text{if } (c > 0) \tilde{a}_1 \text{ else } \tilde{a}_2)} \frac{\text{WF}(\tilde{a}) \text{FV}(\tilde{a}) = \emptyset}{\text{Sync}(\text{spawn } \tilde{a})} \text{Sync}(a)
\end{array}$$

The behaviour \tilde{a} is well-formed when $\text{WF}(\tilde{a})$ can be inferred by the rules above. The free recursion variables of \tilde{a} are denoted by $\text{FV}(\tilde{a})$.

3.3 Behaviour semantics

Verifying concurrent systems of sequential expressions against the policy amounts to a state exploration of the program's abstract model. It is sufficient to track the policy state at the summary boundaries inside of the critical sections only as the type and effect system guarantees non-violation within critical sections. This allows behaviours to be further abstracted by removing resource accesses

$$\beta ::= [] \mid \beta; \tilde{a} \quad \beta_t ::= \tilde{t} \mid \beta_t \mid \beta_t \parallel \tilde{t} \mid t[\beta]$$

$\langle \mathbf{q}; \star; \sigma; \beta_t[\epsilon; \tilde{a}] \rangle$	$\longrightarrow \langle \mathbf{q}; \star; \sigma; \beta_t[\tilde{a}] \rangle$	BHV-SKIP
$\langle \mathbf{q}; \emptyset; \sigma; \beta_t[\text{acq}] \rangle$	$\longrightarrow \langle \mathbf{q}; t; \sigma; \beta_t[\epsilon] \rangle$	BHV-ACQ
$\langle \mathbf{q}; t; \sigma; \beta_t[\text{rel}] \rangle$	$\longrightarrow \langle \mathbf{q}; \emptyset; \sigma; \beta_t[\epsilon] \rangle$	BHV-REL
$\langle \mathbf{q}; \star; \sigma; \beta_t[\tilde{a}_S] \rangle$	$\longrightarrow \langle \mathbf{q}; \star; \sigma; \beta_t[\tilde{a}] \rangle$ if $\mathbf{q} \in S$	
	$\longrightarrow \text{ERROR}$ if $\mathbf{q} \notin S$	BHV-CHECK
$\langle \mathbf{q}_1; \star; \sigma; \beta_t[\tilde{a}_\Psi] \rangle$	$\longrightarrow \langle \mathbf{q}_2; \star; \sigma; \beta_t[\tilde{a}] \rangle$ if \mathbf{q}_1 or $\bar{\mathbf{q}}_1 \in \Psi(\mathbf{q}_2)$	BHV-UPDATE
$\langle \mathbf{q}; \star; \sigma; \beta_t[\text{c?}] \rangle$	$\longrightarrow \langle \mathbf{q}; \star; \sigma; \beta_t[\epsilon] \rangle$ if $\sigma(\text{c}) > 0$	BHV-BLOCK
$\langle \mathbf{q}; \star; \sigma; \beta_t[\text{inc}(\text{c})] \rangle$	$\longrightarrow \langle \mathbf{q}; \star; \sigma + \text{c}; \beta_t[\epsilon] \rangle$	BHV-INC
$\langle \mathbf{q}; \star; \sigma + \text{c}; \beta_t[\text{dec}(\text{c})] \rangle$	$\longrightarrow \langle \mathbf{q}; \star; \sigma; \beta_t[\epsilon] \rangle$	BHV-DEC
$\langle \mathbf{q}; \star; \sigma; \beta_t[\text{spawn } \tilde{a}] \rangle$	$\longrightarrow \langle \mathbf{q}; \star; \sigma; \beta_t[\epsilon] \parallel t'[\tilde{a}] \rangle$ t' fresh	BHV-SPAWN
$\langle \mathbf{q}; \star; \sigma; \beta_t[\mu X.a] \rangle$	$\longrightarrow \langle \mathbf{q}; \star; \sigma; \beta_t[\text{unfold}(\mu X.a)] \rangle$	BHV-UNFOLD
$\langle \mathbf{q}; \star; \sigma; \beta_t[\text{if } (\text{c} > 0) \tilde{a}_1 \text{ else } \tilde{a}_2] \rangle$	$\longrightarrow \langle \mathbf{q}; \star; \sigma; \beta_t[\tilde{a}_i] \rangle$ $i = 1$ if $\sigma(\text{c}) > 0$ else $i = 2$	BHV-CHOICE

and sync effects. We write $[\tilde{a}]$ for these further abstract behaviour effects, defined by $[\text{sync } \tilde{a}] = \tilde{a}$, $[\pi] = \epsilon$. $[\]$ acts homomorphically on all other effects. We allow state exploration of the abstract model by defining a reduction semantics on systems of further abstract effects. The reduction rules are defined on tuples, $\langle \mathbf{q}; \star; \sigma; \tilde{t} \rangle$, where \mathbf{q} is the current state of the policy, \star is the state of the monitor, σ is the signal buffer and \tilde{t} is the abstract system. The only safety check occurs whenever a thread reaches a point of control that has been verified during type checking from states S (BHV-CHECK). The current state \mathbf{q} is updated only by the annotated effect \tilde{a}_Ψ (BHV-UPDATE). The new state may non-deterministically be any \mathbf{q}_2 for which \mathbf{q}_1 is in $\Psi(\mathbf{q}_2)$.

4 Type and effect system

The purpose of the type and effect system is firstly to check that sequential programs do not drive the policy into a dead state and secondly to extract effect behaviours \tilde{a} from expressions. Once we have such effect behaviours for each thread of a concurrent system, we may check further that no policy violation occurs due to concurrency by checking that the *ERROR* state is unreachable by behaviour evaluation on the collection of abstract behaviours $[\tilde{a}]$.

The type and effect system assigns to an expression an annotated simple type τ and a behaviour \tilde{a} . The syntax of types is: $\tau ::= \text{unit} \mid \text{bool} \mid \tau \xrightarrow[\pi S]{\tilde{a}} \tau$. The function type is decorated with the π and S that also decorates the function that is assigned this type, and the latent effect \tilde{a} . A type environment E maps a program's free variables to annotated simple types. Typing takes place with respect to an ambient policy automaton. Type and effect judgements have the form $E; \Psi \vdash e : \tau \ \& \ \tilde{a}$ meaning that e 's function calls are safe with respect to the policy where \tilde{a} is an approximation of e 's runtime behaviour under the assumptions, E and Ψ . When e is protected by the monitor then the judgement means that e can execute safely from each possible current state \mathbf{q} in $\text{dom}(\Psi)$,

$$\begin{array}{c}
E; \Psi \vdash () : \text{unit} \ \& \ \epsilon \quad E; \Psi \vdash x : \tau \ \& \ \epsilon \quad x : \tau \in E \quad E; \Psi \vdash \text{inc}(c) : \text{unit} \ \& \ \text{inc}(c) \\
E; \Psi \vdash \text{true} : \text{bool} \ \& \ \epsilon \quad E; \Psi \vdash \text{false} : \text{bool} \ \& \ \epsilon \quad E; \Psi \vdash \text{dec}(c) : \text{unit} \ \& \ \text{dec}(c) \\
\\
\text{T-FUN} \frac{E, g : \tau_1 \xrightarrow[\pi S]{\mu X. \tilde{a}} \tau_2, x : \tau_1; \Psi \vdash e : \tau_2 \ \& \ \text{unfold}(\mu X. \tilde{a})}{E; \Psi' \vdash \text{fn}_{\pi S} g \ x.e : \tau_1 \xrightarrow[\pi S]{\mu X. \tilde{a}} \tau_2 \ \& \ \epsilon} \Psi = \bigcup \{(\delta(q, \pi), \tilde{q}) \mid q \in S\} \\
\text{dead} \notin \text{dom}(\Psi) \\
\text{T-WAIT} \ E; \Psi \vdash \text{wait}_S c : \text{unit} \ \& \ \text{rel}_{\Psi'}; c?; \text{acq}_S \text{ where } \Psi \subseteq \Psi' \\
\\
\text{T-CHECK} \frac{E; S \cap \Psi \vdash e : \tau \ \& \ \tilde{a}}{E; \Psi \vdash se : \text{unit} \ \& \ \tilde{a}_S} \\
\\
\text{T-SYNC} \frac{E; \Psi \vdash e : \tau \ \& \ \tilde{a}_1}{E; \Psi'' \vdash \text{sync}_S e : \tau \ \& \ \text{sync}(\text{acq}_S; \tilde{a}_1; \text{rel}_{\Psi'})} \text{SIDE}_1: \delta(\Psi, \tilde{a}_1) \subseteq \Psi' \\
\text{SIDE}_2: \Psi = \{(q, q) \mid q \in S\} \\
\\
\text{T-IN-SYNC} \frac{E; \Psi \vdash e : \tau \ \& \ \tilde{a}_1}{E; \Psi \vdash \text{in-sync } e : \tau \ \& \ \text{sync}(\tilde{a}_1; \text{rel}_{\Psi'})} \text{SIDE}_1 \\
\\
\text{T-WAITING} \frac{E; \Psi \vdash e : \tau \ \& \ \tilde{a}_1}{E; \Psi'' \vdash \text{waiting}_S c e : \tau \ \& \ \text{sync}(c?; \text{acq}_S; \tilde{a}_1; \text{rel}_{\Psi'})} \text{SIDE}_1 \ \& \ \text{SIDE}_2 \\
\\
\text{T-SPAWN} \frac{E; \Psi' \vdash e : \tau \ \& \ \tilde{a}}{E; \Psi \vdash \text{spawn } e : \text{unit} \ \& \ \text{spawn } \tilde{a}} \text{in-sync does not occur in } e \\
\\
\text{T-APP} \frac{E; \Psi \vdash e_1 : \tau_2 \xrightarrow[\pi S]{\tilde{a}} \tau_1 \ \& \ \tilde{a}_1 \quad E; \Psi' \vdash e_2 : \tau_2 \ \& \ \tilde{a}_2}{E; \Psi \vdash e_1 e_2 : \tau_1 \ \& \ \tilde{a}_1; \tilde{a}_2; \pi; \tilde{a}_3} \begin{array}{l} \text{SIDE}_1. \ \tilde{a} = \mu X. \tilde{a}'. \\ \text{If } \text{MarkedStates}(\tilde{a}') = \emptyset \\ \text{then } \tilde{a}_3 = \tilde{a} \ \text{or } \tilde{a}_{\lambda q. q} \\ \text{otherwise } \tilde{a}_3 = \tilde{a}_{\Psi''} \ \& \ \text{SIDE}_3. \\ \text{If } \tilde{a}_3 \text{ is of the form } \tilde{a}_{\Psi''' } \\ \text{then } \text{SIDE}_4. \end{array} \\
\\
\text{T-IF} \frac{E; \Psi \vdash e_i : \tau \ \& \ \tilde{a}_i, \ i=1,2}{E; \Psi \vdash \text{if } (c > 0) e_1 \ \text{else } e_2 : \tau \ \& \ \text{if } (c > 0) \tilde{a}_1 \ \text{else } \tilde{a}_2}
\end{array}$$

$$\text{SIDE}_3: \delta(\Psi, \tilde{a}_1; \tilde{a}_2) \subseteq \Psi'' \quad \text{SIDE}_4: \text{dom}(\delta(\Psi, \tilde{a}_1; \tilde{a}_2)) \subseteq S$$

and that the current summary edges are given by $s \longrightarrow q$ for each s and q such that $s \in \Psi(q)$. Values are assigned the empty effect ϵ and can be typed with respect to any Ψ .

The rule T-FUN requires the function body to run safely from $\delta(q, \pi)$ for each q in S . The body is type checked with respect to Ψ which begins a new summary and allows marked state to occur freely in the effect of the function. These marked states in the latent effects act as an indicator to the rule T-APP , that a summary from the calling context is required at application of this function. The check $\text{dead} \notin \text{dom}(\Psi)$ is crucial to avoid policy violation errors.

In the rule T-APP , a function's argument, e_2 , executes after the function expression e_1 terminates, so e_2 must be typed with respect to the states after e_1 ; $\delta(\Psi, \tilde{a}_1)$. The beta reduction is recorded with the effect π . The free marked states of a \tilde{a}' , $\text{MarkedStates}(\tilde{a}')$, denotes the marked states in \tilde{a} which do not

occur beneath a recursion variable binding, μX . The current transaction ends inside the function being applied here, when the function's latent effect contains free marked states. In this case the current summary, Ψ'' , at the point of the beta reduction is annotated on \tilde{a} . When \tilde{a}_3 is of the form $\tilde{a}_{\Psi''}$ then by well-formedness the application must occur beneath sync. In this case the resource access is protected and SIDE_4 requires each possible policy state \mathbf{q} immediately before the beta reduction, to be contained in S since the body of the function is well-typed to run from $\delta(\mathbf{q}, \pi)$ for each $\mathbf{q} \in S$. The rule T-WAIT assigns a wait expression the release effect assuming the current state information Ψ ; followed with a decrement action and then the acquire effect asserting S . Critical sections waiting to enter the monitor are decorated with S asserting that upon entering the critical section the state must be in S . The typing rules T-SYNC and T-WAITING require that a critical section e respects the assertion by type checking e with respect to Ψ with $\text{dom}(\Psi) = S$. When e begins execution the state at the start of the current atomic section for each $\mathbf{q} \in S$ is \mathbf{q} so we have $\mathbf{q} \in \Psi(\mathbf{q})$. All critical sections end by releasing the lock. The states in the release effect at the termination of a critical section are obtained from the states at the beginning of the scope of the critical section e and the entire effect of e (cf. SIDE_1). The remaining typing rules are straightforward.

4.1 Verifying concurrent systems

We now consider the rule for checking a system of threads.

$$\frac{\begin{array}{l} \mathbf{q} \neq \text{dead}. e_i \text{ is in its critical section iff } \star = t_i. \mathbf{s} = \mathbf{q} \text{ whenever } \star = \emptyset. \\ \mathbf{s} = \mathbf{q}' \text{ or } \bar{\mathbf{q}}' \text{ and } \langle \mathbf{q}'; \star; \sigma; t_1[\tilde{a}_1] \parallel \dots \parallel t_n[\tilde{a}_n] \rangle \not\rightarrow^* \text{ERROR} \\ \emptyset; (\mathbf{q}, \mathbf{s}) \vdash e_i : \tau_i \ \& \ \tilde{a}_i \text{ and } \text{WF}(\tilde{a}_i) \text{ for } i \in \{1, \dots, n\} \end{array}}{\langle \mathbf{q}, \mathbf{s} \rangle; \star; \sigma \vdash t_1 e_1 \parallel \dots \parallel t_n e_n}$$

A collection of thread expressions are type checked against the policy resulting in an effect behaviour for each expression. Expressions' monitor usage is then checked by well-formedness of these effects. Once expressions have been individually verified, we can verify their concurrent execution by checking if the interleavings of their behaviours reaches an error. To emphasize the encapsulation of policy state changes into summaries, the state space search is conducted on abstracted behaviours, $[\tilde{a}_i]$.

Note that the rule ensures that thread t_i is evaluating under *in-sync* if and only if thread t_i holds the monitor lock; thus eliminating errors caused by ER_3 .

State \mathbf{q} is the current state of the policy. When thread t_i holds the monitor lock then \mathbf{q}' is the state of the policy from the start of the current summary for t_i . State \mathbf{q}' is therefore the policy state at the boundaries of summaries within atomic sections, and so \mathbf{q}' , and not \mathbf{q} , is used in the initial configuration of the model. We now state the soundness properties of our analysis and a decidability result for a subset of the behaviour expressions. We omit the proofs of these theorems for reasons of space.

Theorem 1 (Subject reduction). *If $\langle \mathbf{q}, \mathbf{s} \rangle; \star; \sigma \vdash T$ and $\langle \mathbf{q}; \star; \sigma; T \rangle \longrightarrow \langle \mathbf{q}'; \star'; \sigma'; T' \rangle$ then $\langle \mathbf{q}', \mathbf{s}' \rangle; \star'; \sigma' \vdash T'$ for some \mathbf{s}' .*

Theorem 2 (Safety). *If $\langle q, s \rangle; \star; \sigma \vdash T$ then $\langle q; \star; \sigma; T \rangle \Downarrow$.*

Theorem 3 (Decidability). *If \tilde{t} is finite state then $\langle q; \star; \sigma; \tilde{t} \rangle \longrightarrow^* ERROR$ is decidable.*

The proof for Subject Reduction follows by induction on the type derivation. Safety follows easily from the definition of the error reductions. Note that Theorem 3 is nontrivial since σ is potentially unbounded. The proof of decidability uses a finite representation of unbounded domain values by introducing \top into the domain of values. A direct corollary of Theorems 1 and 2 is that we can erase the policy state q from the configurations in the reduction rules for expressions, thus avoiding the overhead of tracking the policy state at runtime.

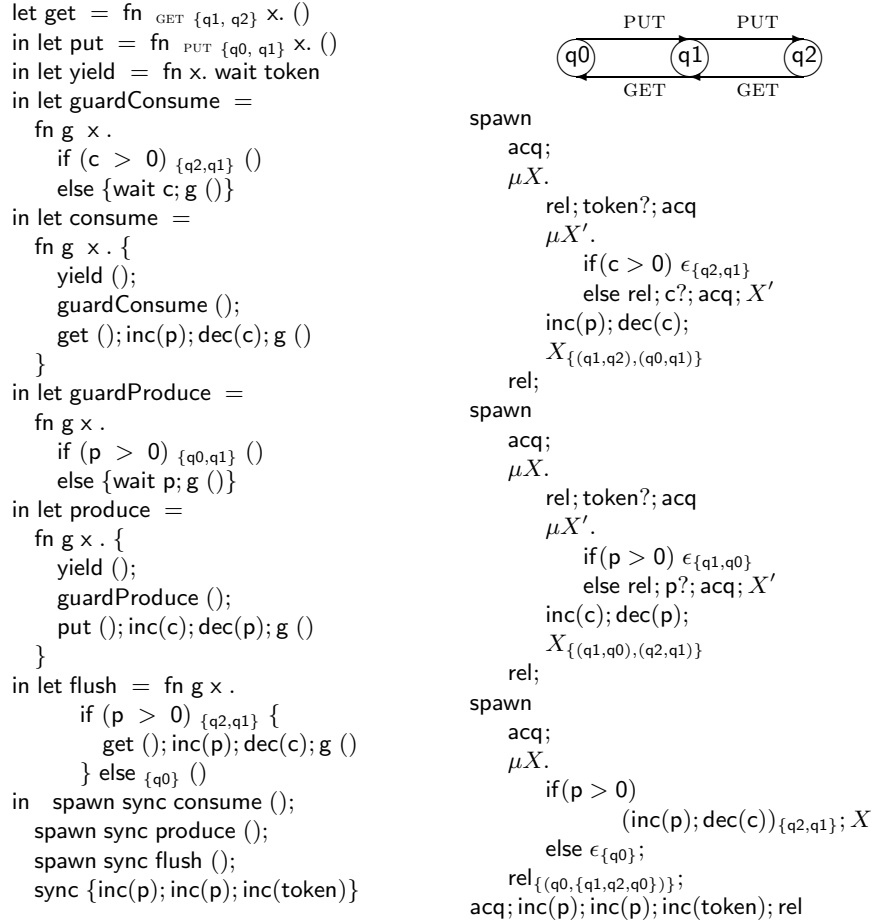


Fig. 2. Standard bounded buffer solution(left), its behaviour model(right).

5 Example

In this section we provide an example generated by an implementation of the analysis presented in this paper. The policy in Figure 4.1 specifies two resources GET and PUT for a buffer with a capacity of two.

The type system accepts the program on the left of figure Figure 4.1 with respect to the buffer policy and generates the behaviour model on the right ¹.

The program implements a standard solution to the bounded buffer problem with one consumer and one producer. The integer variables c and p are used to encode the number of available accesses to the resources GET and PUT.

The consumer loops and calls the function `get` at each iteration. The consumer calls `yield` at the start of the loop. This prevents the consumer from calling `get` more than once in immediate succession, and gives the producer a fair chance to enter the monitor.

The function `guardConsume` returns when $c > 0$, which indicates that GET is available. The producer thread is symmetric.

A third thread repeatedly flushes the buffer in one transaction. The variable p is initialized to 2 and `token` is initialized to 1. The initial default value of c is 0.

The SPIN model checker verified that the Promela translation of the behaviour model, in the initial state q_0 , is free from assertion errors, proving that the program respects the buffer policy.

The state space of the model consists of only the following configuration pairs consisting of the policy state and the variable store,

$$\{(q_0, \{c = 0, p = 2\}), (q_1, \{c = 1, p = 1\}), (q_2, \{c = 2, p = 0\})\}$$

although the state space of the model which includes the control state of each thread is considerably larger ².

Intuitively, the concurrent model respects the policy because it maintains the invariant

$$I : q_i \text{ iff } c = i \text{ and } p = 2 - i$$

whenever the monitor lock is free.

This suggests that there is a more efficient technique to verify this behaviour model which avoids a state space search of full interleavings. Instead we can model check each sequential thread independently, ensuring that whenever a thread enters the monitor assuming I then it also guarantees I at exit to the monitor.

6 Conclusions and further work

We have presented a system for statically checking concurrent programs against an automata based access control mechanism. Our static checks are carried out

¹ Annotations $\Psi = \emptyset$, $\Psi = \lambda q.q$, $S = \{*\}$ and $\pi = \epsilon$ are omitted.

² SPIN used 6 MegaBytes of memory to store the state space of the behaviour model.

in two phases. A type and effect system conducts sequential checks inside atomic sections of the source language and extracts effect behaviour programs. Behaviour effects are improved for model checking by abstracting overall effects of atomic sections on the policy, helping to reduce the number of distinct states per thread. The state changes within atomic sections are encapsulated on effect behaviours by annotations, Ψ , which summarize the policy state within atomic sections. The boundaries of atomic sections, in general, overlap the call stack boundaries of functions. Summaries of atomic sections can therefore be partitioned by function calls.

Areas for further work include automatic inference of the state annotations of our language; compositional reasoning for the model checking; encapsulating the state of integer variables across atomic sections, as we did for the policy state with the Ψ mappings; and scaling the analysis to an object language which prescribes a policy to each object.

An implementation and full details of the static analysis including proofs are available in the first author's thesis [18].

References

1. T. Amtoft, F. Nielson, and H.R Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.
2. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A Model Checker for Concurrent Software. Technical Report MSR-TR-2004-10, Microsoft Research, 2004.
3. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
4. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, first edition, 1984.
5. S. Chaki, S.m K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 45–57. ACM Press, 2002.
6. Y. Cheon and G. Leavens. A runtime assertion checker for the Java modeling language. In *Software Engineering Research and Practice (SERP'02)*, pages 322–328. CSREA Press, 2002.
7. N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder specification language. In *Workshop on Policies for Distributed Systems and Networks (Policy2001)*, Lecture Notes in Computer Science, pages 18–39. Springer-Verlag, 2001.
8. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69. ACM Press, 2001.
9. D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001.
10. S. Godik and T. Moses. eXtensible access control markup language(XACML) version 2.0, 2003.

11. M. Hennessy, J. Rathke, and N. Yoshida. SafeDpi: a language for controlling mobile code. In *Proc. Foundations of Software Science and Computation Structures (FoSSaCS), Barcelona*, Lecture Notes in Computer Science, pages 241–256. Springer-Verlag, 2004.
12. C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
13. A. Igarashi and N. Kobayashi. Resource usage analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 331–342. ACM Press, 2002.
14. Butler Lampson. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton University, 1971.
15. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57. ACM Press, 1988.
16. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 85–97, New York, NY, USA, 1998. ACM Press.
17. G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, Paris, January 1997.
18. Nicholas Nguyen. *Typed Static Analysis for Concurrent, Policy-Based, Resource Access Control*. PhD thesis, Department of Informatics, University of Sussex, 2005.
19. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. Extended abstract in LICS '93.
20. S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–255, New York, NY, USA, 2004. ACM Press.
21. M. Tofte and Jean-Pierre Talpin. Region-based memory management. In *Information and Computation*, volume 132, pages 109–176. Academic Press, 1997.
22. D. Walker. A type system for expressive security policies. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 254–267. ACM Press, 2000.
23. D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *Proceedings of 1998 IEEE Symposium on Security and Privacy*, pages 52–63, 1998.