Securing the Future — an Information Flow Analysis of a Distributed OO Language *

Martin Pettai[1,2] and Peeter Laud[1,2]

[1] University of Tartu
[2] Cybernetica AS

Abstract. We present an information-flow type system for a distributed object-oriented language with active objects, asynchronous method calls and futures. The variables of the program are classified as high and low. We allow while cycles with high guards to be used but only if they are not followed (directly or through synchronization) by an assignment to a low variable. To ensure the security of synchronization, we use a high and a low lock for each concurrent object group (cog). In some cases, we must allow a high lock held by one task to be overtaken by another, if the former is about to make a low side effect but the latter cannot make any low side effects. This is necessary to prevent synchronization depending on high variables from influencing the order of low side effects in different cogs. We prove a non-interference result for our type system.

1 Introduction

The question of information security arises when the inputs and outputs of a program are partitioned into different security classes. In this case we want the high-security inputs not inappropriately influence the low-security outputs and other behaviour observable at low clearance. The strongest such property is non-interference [9] stating that there is no influence at all; or that variations in the high-security inputs do not change the observations at the low level.

Over the years, static analyses, typically type systems for verifying secure information flow have been proposed for programs written in many kinds of programming languages and paradigms — imperative or functional, sequential or parallel, etc. Each new construct in the language can have a profound effect on the information flows the programs may have. With the spread of distributed computing and multi-core processors, concurrent object-oriented programming is gaining mindshare. The languages supporting this paradigm emphasize the greater independence of objects and various methods of communication between

* The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 231620 (HATS — Highly Adaptable and Trustworthy Software using Formal Models), from Estonian Science Foundation through grants no. 7543 and 8124, and from the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS.

the objects and the concurrently running tasks. The effect these constructs have on the information flow has not yet been thoroughly investigated. On the one hand, the communication primitives are prone to introducing information leaks. On the other hand, the explicit independence of objects and their groups can provide clear evidence that certain flows are missing.

In this paper, we investigate a particular concurrent object-oriented language, related to Creol [12] and JCoBoxes [21]. The language, its syntax and semantics is almost the same as the concurrent OO sublanguage of the *core Abstract Behavioural Specification Language (ABS)* [10] and shall henceforth be named as this. At the core of this language lies the notion of a *concurrent object group* (cog). Different cogs run concurrently and independently of each other and communicate only via asynchronous method calls. When placing such a call, a *future* [8] (a placeholder for the eventually available return value) is immediately returned. A future admits certain operations for checking the presence of and reading the return value. Inside a cog, there may also be several running tasks sharing some common state (the fields of the object). In contrast, these tasks are scheduled cooperatively, such that there is always just a single active task per cog.

In this paper we propose a type system for checking the non-interference in programs written in ABS. For eliminating certain information flows, and for simplifying the checks we will fix or adjust certain details of the language in a manner that can be seen as non-essential for its purposes (specifying concurrent systems). Compared to [10], our language has a more fine-grained system of locks for controlling which task is currently running inside a cog. We have also restricted the scheduler for non-preemptive tasks, such that information flow properties are easier to enforce. The specification of scheduling decisions is made harder by the necessity to not introduce deadlocks into the program that were not there before. On the other hand, different cogs are running in a truly parallel fashion, scheduled nondeterministically.

We will introduce the syntax and semantics of ABS in Sec. 2. While describing the syntax, we will already introduce security types and annotations that form the basis for defining non-interference. In Sec. 3 we introduce our type system for secure information flow, and state the properties satisfied by well-typed programs. We review the related work in Sec. 4 and discuss our results in Sec. 5.

2 The Programming Language

2.1 Syntax

Our programming language is a simplified version of ABS. Its (abstract) syntax is given in Fig. 1. The notation $\overline{X}$ denotes a sequence of X-s. Several constructs in the syntax are annotated by security levels. These do not have to be provided by the programmer, as they can be inferred automatically during type checking.

Let us explain the language constructs related to distributed execution. $e_p!m(\overline{e_p})$ denotes the *asynchronous* call of the method m. The call immediately

$x \mid n \mid o \mid b \mid f$	local variable | task | object | cog | field name
$Pr ::= \overline{Cl}\ B$	program
$Cl ::= \mathsf{class}\ C\{\overline{T\,f}\ \overline{M}\}$	class definition
$M ::= (m : (l, \overline{T}) \overset{l[,i]}{\to} \mathsf{Cmd}^l(T))(\overline{x})\ B$	method definition
$B ::= \{\overline{T\,x}\ s; x\}$	method body
$v ::= x \mid \mathsf{this} \mid \mathsf{this}.f$	variable
$i ::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots$	integer
$e ::= e_p \mid e_s$	expression
$e_p ::= v \mid \mathsf{null} \mid i \mid e_p = e_p$	pure expression
$e_s ::= e_p!_l m(\overline{e_p}) \mid e_p.\mathsf{get}_l \mid \mathsf{new}\ C \mid \mathsf{new\ cog}\ C$	expression with side effects
$s ::= v := e \mid e \mid \mathsf{skip} \mid \mathsf{suspend}_l \mid \mathsf{await}_l\ g$ $\mid \mathsf{if}\ (e_p)\ s\ \mathsf{else}\ s \mid \mathsf{while}_l\ (e_p)\ s \mid s; s$	statement
$g ::= v?$	guard
$l ::= L \mid H$	security level
$\ell ::= l \mid i$	security level or integer
$T ::= \mathsf{Int}_l \mid C_l \mid \mathsf{Fut}_l^{\ell}(T) \mid \mathsf{Guard}_l^{\ell}$	security type

Fig. 1. Syntax

returns a future. The get-construct is used to read the value of that future, if it is available. If not, then get blocks. The suspend-statement suspends the current thread, it is used for non-preemptive scheduling inside a cog. The statement await g suspends until the guard $g = v?$ becomes true, which happens when the future v obtains a value.

2.2 Operational Semantics

We first define run-time configurations. The program at run time is a set of cogs (concurrent object groups), each of which contains a set of objects. Each object is related to zero or more tasks. The run-time configurations are as follows:

$$P ::= b[n_1, n_2] \mid o[b, C, \sigma] \mid n\,\langle b, o, \sigma, s\rangle \mid P \parallel P$$

Each cog is represented by its identifier b and the state of its locks. Each cog has two locks—the low lock, which is owned by task n_1 (or is free if $n_1 = \bot$), and the high lock, which is owned by task n_2 (or is free if $n_2 = \bot$).

Each object is represented by its identifier o, its cog b, its class C, and its state σ (the values of its fields). Each task is represented by its identifier n, its cog b, its object o, the statement s that is yet to be executed in this task, and its state σ (the values of its local variables).

The run-time syntax will have some additional constructs:

$$e_p ::= \dots \mid n \mid o \qquad s ::= \dots \mid \mathsf{grab}_l \mid \mathsf{release}_l \qquad a ::= \mathsf{null} \mid i \mid n \mid o$$

Thus task and object identifiers can be used (these result from evaluating other expressions) and we will use a to denote fully reduced expressions (i.e. constants): Also separate statements are introduced for grabbing and releasing locks (used for executing suspend and when starting and terminating tasks).

The initial configuration for the program $\overline{Cl}\ \{\overline{T\,x}\ s; x_0\}$ will be

$$b_0[n_0, n_0] \parallel n_0\ \langle b_0, \mathsf{null}, \sigma, s; \mathsf{release}_L; x_0\rangle$$

i.e. an initial cog b_0 will be created for the task n_0 executing the main method. This task will have both locks initially and the statement $\mathsf{release}_L$ is added to release the locks before the task terminates. This task is the only task that is not tied to an object (all tasks created later will be tied to some object). The variable x_0 (which must have type Int_L) will contain the return value of the program. Input can be given to the program through the initial values of the variables in σ. These variables must be declared in the body of the main method.

Now we can give the reduction rules (including the necessary reduction contexts) in Fig. 2. Again, some explanations are in order. The commands grab and release manipulate the locks of a cog. Suspending a task is equivalent to releasing a lock and then trying to grab it again. A method body that starts with a grab is currently suspended. It is possible to perform either a low or a high suspend. When a task has performed a high suspend, then only other high-suspended tasks can continue.

An asynchronous call (acall) creates a new task in the cog containing the receiver of the call. The new task is initially suspended. The name of the new task is used as the future.

A while-loop suspends after each iteration. Hence an infinite loop cannot stop the computation in the entire cog and cause information flows through non-termination in such manner. The semantics of the await-command is straightforward, except for the rule (await_3). It is used to avoid certain deadlocks. See Sec. 3.1 for the definition of low and high-low tasks and further discussion. Basically, rule (await_3) allows the task n' to preemptively start running (and suspend the task n_1) if its final value is being waited for. In such manner, the possible non-termination of task n_1 cannot affect the termination behaviour of n (the high-low task n' always terminates).

3 Type System for Non-Interference

3.1 Types

The types in the type system are the following:

$$T ::= \mathsf{Int}_l \mid C_l \mid \mathsf{Fut}_l^\ell(T) \mid \mathsf{Guard}_l^\ell \mid \mathsf{Exp}^l(T) \mid \mathsf{Cmd}^l \mid \mathsf{Cmd}^l(T) \mid (l, \overline{T}) \stackrel{l[,i]}{\rightarrow} \mathsf{Cmd}^l(T)$$

$$R_1[e] ::= x := e \mid \mathsf{this}.f := e$$

$$R_2[e] ::= R_1[e] | \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2 | R_1[e.\mathsf{get}_l] | R_1[e!_l m(\overline{e'})] | R_1[e'!_l m(\overline{e_1}\, e\, \overline{e_2})] | R_2[e = e'] | R_2[e' = e]$$

$$\frac{n' \text{ fresh} \qquad \mathrm{body}(m) = s(\bar{x}); x' \qquad s_{task} = \mathsf{grab}_l; s[\bar{a}/\bar{x}]; \mathsf{release}_l; x'}{o'[b', C, \sigma'] \parallel n\,\langle b, o, \sigma, R_1[o'!_l m(\bar{a})]; s\rangle \rightsquigarrow o'[b', C, \sigma'] \parallel n\,\langle b, o, \sigma, R_1[n']; s\rangle \parallel n'\,\langle b', o', \sigma_{init}, s_{task}\rangle}\ (\mathsf{acall})$$

$$\frac{o' \text{ fresh}}{n\,\langle b, o, \sigma, R_1[\mathsf{new}\ C]; s\rangle \rightsquigarrow n\,\langle b, o, \sigma, R_1[o']; s\rangle \parallel o'[b, C, \sigma_{init}]}\ (\mathsf{new})$$

$$\frac{b' \text{ fresh} \qquad o' \text{ fresh}}{n\,\langle b, o, \sigma, R_1[\mathsf{new\ cog}\ C]; s\rangle \rightsquigarrow n\,\langle b, o, \sigma, R_1[o']; s\rangle \parallel b'[\bot, \bot] \parallel o'[b', C, \sigma_{init}]}\ (\mathsf{newcog})$$

$$\frac{}{n\,\langle b, o, \sigma', R_1[n'.\mathsf{get}_l]; s\rangle \parallel n'\,\langle b', o', \sigma, x\rangle \rightsquigarrow n\,\langle b, o, \sigma', R_1[\sigma(x)]; s\rangle \parallel n'\,\langle b', o', \sigma, x\rangle}\ (\mathsf{get}_1)$$

$$\frac{}{n\,\langle b, o, \sigma', R_1[n'.\mathsf{get}_l]; s\rangle \parallel n'\,\langle b', o', \sigma, s'; x\rangle \rightsquigarrow n\,\langle b, o, \sigma', \mathsf{await}_l(n'?); R_1[n'.\mathsf{get}_l]; s\rangle \parallel n'\,\langle b', o', \sigma, s'; x\rangle}\ (\mathsf{get}_2)$$

$$\frac{}{n\,\langle b, o, \sigma, R_2[x]; s\rangle \rightsquigarrow n\,\langle b, o, \sigma, R_2[\sigma(x)]; s\rangle}\ (\mathsf{var})$$

$$\frac{}{o[b, C, \sigma] \parallel n\,\langle b, o, \sigma', R_2[\mathsf{this}.f]; s\rangle \rightsquigarrow o[b, C, \sigma] \parallel n\,\langle b, o, \sigma', R_2[\sigma(f)]; s\rangle}\ (\mathsf{field})$$

$$\frac{}{n\,\langle b, o, \sigma, a; s\rangle \rightsquigarrow n\,\langle b, o, \sigma, s\rangle}\ (\mathsf{dummyexpr})$$

$$\frac{}{n\,\langle b, o, \sigma, x := a; s\rangle \rightsquigarrow n\,\langle b, o, \sigma[x \mapsto a], s\rangle}\ (\mathsf{assignvar})$$

$$\frac{}{o[b, C, \sigma] \parallel n\,\langle b, o, \sigma', \mathsf{this}.f := a; s\rangle \rightsquigarrow o[b, C, \sigma[f \mapsto a]] \parallel n\,\langle b, o, \sigma', s\rangle}\ (\mathsf{assignfield})$$

$$\frac{}{n\,\langle b, o, \sigma, \mathsf{skip}; s\rangle \rightsquigarrow n\,\langle b, o, \sigma, s\rangle}\ (\mathsf{skip})$$

$$\frac{}{n\,\langle b, o, \sigma, \mathsf{suspend}_l; s\rangle \rightsquigarrow n\,\langle b, o, \sigma, \mathsf{release}_l; \mathsf{grab}_l; s\rangle}\ (\mathsf{suspend})$$

$$\frac{}{b[\bot, \bot] \parallel n\,\langle b, o, \sigma, \mathsf{grab}_L; s\rangle \rightsquigarrow b[n, n] \parallel n\,\langle b, o, \sigma, s\rangle}\ (\mathsf{grab_L})$$

$$\frac{}{b[n', \bot] \parallel n\,\langle b, o, \sigma, \mathsf{grab}_H; s\rangle \rightsquigarrow b[n', n] \parallel n\,\langle b, o, \sigma, s\rangle}\ (\mathsf{grab_H})$$

$$\frac{}{b[n, n] \parallel n\,\langle b, o, \sigma, \mathsf{release}_L; s\rangle \rightsquigarrow b[\bot, \bot] \parallel n\,\langle b, o, \sigma, s\rangle}\ (\mathsf{release_L})$$

$$\frac{}{b[n', n] \parallel n\,\langle b, o, \sigma, \mathsf{release}_H; s\rangle \rightsquigarrow b[n', \bot] \parallel n\,\langle b, o, \sigma, s\rangle}\ (\mathsf{release_H})$$

$$\frac{i \neq 0}{n\,\langle b, o, \sigma, \mathsf{if}\ (i)\ s_1\ \mathsf{else}\ s_2; s\rangle \rightsquigarrow n\,\langle b, o, \sigma, s_1; s\rangle}\ (\mathsf{if}_+)$$

$$\frac{}{n\,\langle b, o, \sigma, \mathsf{if}\ (0)\ s_1\ \mathsf{else}\ s_2; s\rangle \rightsquigarrow n\,\langle b, o, \sigma, s_2; s\rangle}\ (\mathsf{if}_-)$$

$$\frac{}{n\,\langle b, o, \sigma, \mathsf{while}_l\ (e)\ s_1; s_2\rangle \rightsquigarrow n\,\langle b, o, \sigma, \mathsf{if}\ (e)\ (s_1; \mathsf{suspend}_l; \mathsf{while}_l\ (e)\ s_1)\ \mathsf{else\ skip}; s_2\rangle}\ (\mathsf{while})$$

$$\frac{}{n\,\langle b, o, \sigma', \mathsf{await}_l(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, x\rangle \rightsquigarrow n\,\langle b, o, \sigma', s\rangle \parallel n'\,\langle b', o', \sigma, x\rangle}\ (\mathsf{await}_1)$$

$$\frac{}{\begin{array}{c} n\,\langle b, o, \sigma', \mathsf{await}_l(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, s'; x\rangle \rightsquigarrow \\ \rightsquigarrow n\,\langle b, o, \sigma', \mathsf{suspend}_l; \mathsf{await}_l(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, s'; x\rangle \end{array}}\ (\mathsf{await}_2)$$

$$\frac{\text{the next step of } s_1 \text{ is low and the task } n' \text{ is high-low}}{\begin{array}{c} n\,\langle b, o, \sigma', \mathsf{await}_H(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, \mathsf{grab}_H; s'; x\rangle \parallel n_1\,\langle b', o_1, \sigma_1, s_1\rangle \parallel b'[n_1, n_1] \rightsquigarrow \\ \rightsquigarrow n\,\langle b, o, \sigma', \mathsf{suspend}_H; \mathsf{await}_H(n'?); s\rangle \parallel n'\,\langle b', o', \sigma, s'; x\rangle \parallel n_1\,\langle b', o_1, \sigma_1, \mathsf{grab}_H; s_1\rangle \parallel b'[n_1, n'] \end{array}}\ (\mathsf{await}_3)$$

Fig. 2. Reduction rules

Thus we can have integers, objects of class C, futures, guards, possibly non-terminating expressions, commands, commands (method bodies) returning a value, and methods. Here the subscript represents the security level of the value. For integers and objects, this corresponds to the upper bound on the security levels of the inputs that may have affected this value. For futures and guards, this is the upper bound on the (control flow) information that may affect which task this future is referring to. The security level on top of the arrow of the method type corresponds to the minimum level of side effects this method is allowed to perform. If this is high, then the side effects of this method do not affect the low part of the computation. The level l_0 in the method type denotes the security level of the receiver of the method call (i.e., this-argument). The superscripts on the types are the upper bound on information that may affect whether this future, guard, expression, or command eventually returns a value or terminates. If this information is high, then the effects of any computation that follows are high, too.

We also define the security level corresponding to those security types that can be types of variables:

$$level(C_l) = l \qquad level(\mathsf{Int}_l) = l \qquad level(\mathsf{Fut}_l^{\ell'}(T)) = l \qquad level(\mathsf{Guard}_l^{\ell'}) = l$$

Thus $level(T)$ is the maximum context level where assignments to variables of type T are allowed.

The typing rules are given in Figures 3 and 4. The general shape of the

$$l \leq l \qquad L \leq H \qquad \frac{l_2 \leq l_1 \qquad \ell_3 \leq \ell_4}{\mathsf{Guard}_{l_1}^{\ell_3} \leq \mathsf{Guard}_{l_2}^{\ell_4}} \qquad \frac{l_2 \leq l_1 \qquad \ell_3 \leq \ell_4 \qquad T_5 \leq T_6}{\mathsf{Fut}_{l_1}^{\ell_3}(T_5) \leq \mathsf{Fut}_{l_2}^{\ell_4}(T_6)}$$

$$\mathsf{Guard}_H^i \leq \mathsf{Guard}_H^L \qquad \frac{l_1 \leq l_2}{C_{l_1} \leq C_{l_2}} \qquad \frac{l_1 \leq l_2}{\mathsf{Int}_{l_1} \leq \mathsf{Int}_{l_2}} \qquad \frac{\gamma, l \vdash e : T}{\gamma, l \vdash e : \mathsf{Exp}^L(T)}$$

$$\frac{\gamma, l \vdash e : T_1 \qquad T_1 \leq T_2}{\gamma, l \vdash e : T_2} \qquad \frac{\gamma, l \vdash s : \mathsf{Cmd}^{l_1} \qquad l_1 \leq l_2}{\gamma, l \vdash s : \mathsf{Cmd}^{l_2}} \qquad \frac{\gamma, l_1 \vdash s : \mathsf{Cmd}^l \qquad l_1 \geq l_2}{\gamma, l_2 \vdash s : \mathsf{Cmd}^l}$$

Fig. 3. Subtyping rules

typing rules is $\gamma, l \vdash X : T$, where γ is the typing context giving the types of local variables, fields, and methods, l is the current *security context* upper bounding the information that may have affected whether the execution reaches the current program point, X is a typable quantity and T is its type. For typing methods, there is no security context. Considering the meaning of sub- and superscripts in the types, the rules in Figures 3 and 4 should be rather straightforward. A program Pr is well typed if $\vdash Pr : ok$ is derivable.

We also allow an integer i to be added to the security level of the context. This is used to guarantee termination for methods (corresponding to high-low tasks in Def. 2) where the security level of the context is higher than the security level of termination and thus while cycles are forbidden. Cycles could still occur

$$\frac{\begin{array}{c}(\forall i)\ Cl_i = \mathsf{class}\ C_i\{T_{i1}\,f_{i1}\ldots T_{ir_i}\,f_{ir_i}\ M_{i1}\ldots M_{ik_i}\} \qquad (\forall i,j)\ M_{ij} = (m_{ij} : T'_{ij})(\overline{x_{ij}})\ B_{ij} \\ \gamma = \{C_i.f_{ij} \mapsto T_{ij} \mid 1\le i\le n, 1\le j\le r_i\} \cup \{C_i.m_{ij}\mapsto T'_{ij} \mid 1\le i\le n, 1\le j\le k_i\} \\ (\forall i,j)\ \gamma \vdash M_{ij} : T'_{ij} \qquad B = \{\overline{T\ x}\ s\} \qquad \gamma, \overline{x:T}, L \vdash s : \mathsf{Cmd}^L(\mathsf{Int}_L)\end{array}}{\vdash Cl_1\ldots Cl_n\ B : ok}\ (\mathsf{Prog})$$

$$\frac{\gamma,\overline{x:T},\mathsf{this}:C_{l_0}, l \vdash s : \mathsf{Cmd}^{l_1}(T') \qquad l = l_1}{\gamma \vdash (m : (l_0,\overline{T}) \xrightarrow{l} \mathsf{Cmd}^{l_1}(T'))(\overline{x})\ \{\overline{T\ x}\ s\} : (l_0,\overline{T}) \xrightarrow{l} \mathsf{Cmd}^{l_1}(T')}\ (\mathsf{Method}_1)$$

$$\frac{\gamma,\overline{x:T},\mathsf{this}:C_{l_0}, l, i \vdash s : \mathsf{Cmd}^{l_1}(T') \qquad l > l_1 \qquad i > 0}{\gamma \vdash (m : (l_0,\overline{T}) \xrightarrow{l,i} \mathsf{Cmd}^{l_1}(T'))(\overline{x})\ \{\overline{T\ x}\ s\} : (l_0,\overline{T}) \xrightarrow{l,i} \mathsf{Cmd}^{l_1}(T')}\ (\mathsf{Method}_2)$$

$$\frac{\gamma(x) = T}{\gamma, l \vdash x : T}\ (\mathsf{Var}) \qquad \frac{\gamma(C.f) \le T \qquad T \in \{C'_l, \mathsf{Int}_l, \mathsf{Fut}_l^{\ell_1}(T')\} \qquad \gamma, l_2 \vdash \mathsf{this} : C_l}{\gamma, l_2 \vdash \mathsf{this}.f : T}\ (\mathsf{Field})$$

$$\frac{}{\gamma, l \vdash \mathsf{null} : C_L}\ (\mathsf{Null}) \qquad \frac{}{\gamma, l \vdash i : \mathsf{Int}_L}\ (\mathsf{Int})$$

$$\frac{\gamma, l \vdash e : C_{l_0} \qquad \gamma, l \vdash \overline{e} : \overline{T} \qquad \gamma(C.m) = l_0, \overline{T} \xrightarrow{l} \mathsf{Cmd}^{l_1}(T_2) \qquad l_0 \ge l \qquad \overline{T} \ge l \qquad l_1 = l}{\gamma, l \vdash e!_l m(\overline{e}) : \mathsf{Fut}_l^{l_1}(l \vee l_1 \vee T_2)}\ (\mathsf{ACall}_1)$$

$$\frac{\gamma, l \vdash e : C_{l_0} \qquad \gamma, l \vdash \overline{e} : \overline{T} \qquad \gamma(C.m) = l_0, \overline{T} \xrightarrow{l,i} \mathsf{Cmd}^{l_1}(T_2) \qquad l_0 \ge l \qquad \overline{T} \ge l \qquad l_1 < l}{\gamma, l \vdash e!_l m(\overline{e}) : \mathsf{Fut}_l^{i}(l \vee l_1 \vee T_2)}\ (\mathsf{ACall}_2)$$

$$\frac{\gamma, l \vdash e : \mathsf{Fut}_l^{l_1}(T)}{\gamma, l \vdash e.\mathsf{get}_l : \mathsf{Exp}^{l_1}(T)}\ (\mathsf{Get}_1) \qquad \frac{\gamma, l, i \vdash e : \mathsf{Fut}_l^{i_1}(T) \qquad i_1 < i}{\gamma, l, i \vdash e.\mathsf{get}_l : \mathsf{Exp}^{L}(T)}\ (\mathsf{Get}_2)$$

$$\frac{}{\gamma, L \vdash \mathsf{new}\ C : C_L}\ (\mathsf{New}) \qquad \frac{}{\gamma, L \vdash \mathsf{new\ cog}\ C : C_L}\ (\mathsf{NewCog})$$

$$\frac{\gamma, l \vdash e : \mathsf{Exp}^{l_2}(T)}{\gamma, l \vdash e : \mathsf{Cmd}^{l_2}}\ (\mathsf{DummyExpr}) \qquad \frac{}{\gamma, l \vdash \mathsf{skip} : \mathsf{Cmd}^{L}}\ (\mathsf{Skip})$$

$$\frac{\gamma(x) = T \qquad level(T) = l \qquad \gamma, l \vdash e : \mathsf{Exp}^{l_2}(T)}{\gamma, l \vdash x := e : \mathsf{Cmd}^{l_2}}\ (\mathsf{AssignVar})$$

$$\frac{\gamma(C.f) = T \qquad level(T) = l \qquad \gamma, l \vdash e : \mathsf{Exp}^{l_2}(T) \qquad \gamma, l \vdash \mathsf{this} : C_l}{\gamma, l \vdash \mathsf{this}.f := e : \mathsf{Cmd}^{l_2}}\ (\mathsf{AssignField})$$

$$\frac{\gamma, l \vdash e : \mathsf{Guard}_l^{l_1}}{\gamma, l \vdash \mathsf{await}_l(e) : \mathsf{Cmd}^{l_1}}\ (\mathsf{Await}_1) \qquad \frac{\gamma, l, i \vdash e : \mathsf{Guard}_l^{i_1} \qquad i_1 < i}{\gamma, l, i \vdash \mathsf{await}_l(e) : \mathsf{Cmd}^{L}}\ (\mathsf{Await}_2)$$

$$\frac{}{\gamma, l \vdash \mathsf{grab}_l : \mathsf{Cmd}^{L}}\ (\mathsf{Grab}) \qquad \frac{}{\gamma, l \vdash \mathsf{release}_l : \mathsf{Cmd}^{L}}\ (\mathsf{Release}) \qquad \frac{}{\gamma, l \vdash \mathsf{suspend}_l : \mathsf{Cmd}^{L}}\ (\mathsf{Suspend})$$

$$\frac{\gamma, l \vdash e : \mathsf{Int}_l \qquad \gamma, l \vdash s_1 : \mathsf{Cmd}^{l_1} \qquad \gamma, l \vdash s_2 : \mathsf{Cmd}^{l_1}}{\gamma, l \vdash \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2 : \mathsf{Cmd}^{l_1}}\ (\mathsf{If}) \qquad \frac{\gamma, l \vdash e : \mathsf{Int}_l \qquad \gamma, l \vdash s : \mathsf{Cmd}^{l}}{\gamma, l \vdash \mathsf{while}_l\ (e)\ s : \mathsf{Cmd}^{l}}\ (\mathsf{While})$$

$$\frac{\gamma, l \vdash s_1 : \mathsf{Cmd}^{l_1} \qquad \gamma, l \vee l_1 \vdash s_2 : \mathsf{Cmd}^{l_2}}{\gamma, l \vdash s_1; s_2 : \mathsf{Cmd}^{l_1 \vee l_2}}\ (\mathsf{Seq}_1)$$

$$\frac{\gamma, l \vdash s_1 : \mathsf{Cmd}^{l_1} \qquad \gamma, l \vee l_1 \vdash s_2 : \mathsf{Cmd}^{l_2}(T)}{\gamma, l \vdash s_1; s_2 : \mathsf{Cmd}^{l_1 \vee l_2}(T)}\ (\mathsf{Seq}_2)$$

$$\frac{\gamma, l \vdash x : T \qquad level(T) = l}{\gamma, l \vdash x : \mathsf{Cmd}^{L}(T)}\ (\mathsf{ReturnVar}) \qquad \frac{\gamma, l' \vdash e : \mathsf{Fut}_l^{l_1}(T)}{\gamma, l' \vdash e? : \mathsf{Guard}_l^{l_1}}\ (\mathsf{Guard})$$

Fig. 4. Type rules

through cycles in the await graph and to disallow this, each of these methods has a positive integer $i > 0$ and can only await after a task with a smaller integer. This makes the await graph of high-low tasks acyclic. To achieve this, we have some typing rules of the form $\gamma, l, i \vdash X : T$. The integer i can also be assimilated with γ. For example, a rule

$$\frac{\gamma, l, i \vdash s_1 : \mathsf{Cmd}^{l_1} \qquad \gamma, l \vee l_1, i \vdash s_2 : \mathsf{Cmd}^{l_2}}{\gamma, l, i \vdash s_1; s_2 : \mathsf{Cmd}^{l_1 \vee l_2}}$$

is considered a special case of the rule (Seq_1) and thus is not given separately. The integer i is also used (instead of L) in the superscript of the futures and guards of high-low tasks.

By the next definition, we can now distinguish high and low reduction steps, depending on whether the reduced statement is typable in high context or not.

Definition 1. *Let the statement s have the form $s_1; s_2$ where s_1 is not a sequential composition (because of associativity of the sequential composition operator, a statement always has either this form or the form x (a single variable, which cannot be further reduced)). We call the next reduction step of s a* high step *if $\gamma, H \vdash s_1 : \mathit{Cmd}^l$ and a* low step *otherwise.*

The next definition allows to also distinguish high and low tasks. The previous and the next definition are used in the (await_3) rule in Fig. 2.

Definition 2. *We call a task $n \langle b, o, \sigma, s; x \rangle$ a* high task *if $\gamma, H \vdash s : \mathit{Cmd}^l$ and a* low task *otherwise. The high tasks are further distinguished: if $\gamma, H \vdash s : \mathit{Cmd}^L$ then it is a* high-low task *and otherwise it is a* high-high task*.*

A high task can only make high steps, but a low task can make both high and low steps. A high-high task can contain only high cycles (cycles with a high guard) because low cycles are not allowed in high context. A high-low task cannot contain any cycles (because at most low guards are allowed but high context requires at least high guards). We have the following lemma.

Lemma 1. *A low task cannot contain high* while *cycles. A high-low task cannot contain any* while *cycles.*

The restriction on the use of high while cycles is modeled after the restriction in [22]. Thus no low steps can follow a high while cycle in the same task. This restriction is checked in the rules (Seq_1) and (Seq_2). Because a low task must eventually release both locks, which is a low step, a low task cannot contain high while cycles at all. We extend the same restriction to await cycles. Thus a low task cannot await after a task that is allowed to make high cycles.

In our language, the scheduler of a cog cannot switch to a different task before the current task releases the high lock (or both locks). This can be done explicitly using suspend, but it is also done implicitly after each iteration of a while or await cycle. In contrast, in [22], by default the scheduler can switch tasks at any time, this can be disallowed by wrapping a sequence of commands in a protect construct. The protect construct is not allowed to contain cycles. This restriction corresponds to our implicit suspend after each iteration of a cycle.

Because our language allows more than one cog, there can be several low tasks running in parallel (at most one in each cog). This can create a situation where a low task n_1 in one cog (b_1) is in high context and awaits for a high task n_2 in another cog (b_2) but the high lock of cog b_2 is held by a low task n_3 in cog b_2. Thus the task n_1 cannot make the next low step before the task n_2 terminates, which cannot happen before the task n_3 releases the high lock but n_3 may make some low steps before it releases the lock. Thus it may depend on the high variables in n_1 whether low steps must be made in n_3 before the next low step in n_1 or not. Thus the low steps in n_3 are essentially in high context. To prevent this indirect information flow, we allow the task n_2 to overtake the high lock from n_3 in this situation. This means that n_3 is not required to make low steps before n_1 does, no matter what the values of high variables in n_1 are. This is handled by the reduction rule (await_3).

3.2 Non-interference

We first define the low-equivalence relation in Fig. 5. Here we assume (w.l.o.g.) that all variables in the program have globally unique names. Thus we can use a single type context γ instead of separate type contexts for each task.

$$\frac{\gamma, l \vdash s : \mathsf{Cmd}^H}{s \sim_\gamma s} \qquad \frac{\gamma, H \vdash s : \mathsf{Cmd}^H \qquad \gamma, H \vdash s' : \mathsf{Cmd}^H}{s \sim_\gamma s'}$$

$$\frac{\gamma, l \vdash s : \mathsf{Cmd}^H(T)}{s \sim_\gamma s} \qquad \frac{\gamma, H \vdash s : \mathsf{Cmd}^H(T) \qquad \gamma, H \vdash s' : \mathsf{Cmd}^H(T)}{s \sim_\gamma s'}$$

$$\frac{\gamma, H \vdash s_1 : \mathsf{Cmd}^H \qquad s_2 \sim_\gamma s_2'}{s_1; s_2 \sim_\gamma s_2'} \qquad \frac{\gamma, H \vdash s_1 : \mathsf{Cmd}^H \qquad s_2 \sim_\gamma s_2'}{s_2 \sim_\gamma s_1; s_2'} \qquad \frac{s_2 \sim_\gamma s_2'}{s_1; s_2 \sim_\gamma s_1; s_2'}$$

$$\sigma \sim_\gamma \sigma' \equiv dom(\sigma) = dom(\sigma') \land \forall v \in dom(\sigma).\, level(\gamma(v)) = L \Rightarrow \sigma(v) = \sigma'(v)$$

$$b[n_1, n_2] \sim_\gamma b[n_1, n_2']$$

$$\frac{\sigma \sim_\gamma \sigma'}{o[b, C, \sigma] \sim_\gamma o[b, C, \sigma']} \qquad \frac{\sigma \sim_\gamma \sigma' \qquad s \sim_\gamma s'}{n\,\langle b, o, \sigma, s\rangle \sim_\gamma n\,\langle b, o, \sigma', s'\rangle} \qquad \frac{P_1 \sim_\gamma P_1' \qquad P_2 \sim_\gamma P_2'}{P_1 \parallel P_2 \sim_\gamma P_1' \parallel P_2'}$$

$$\frac{\gamma, H \vdash s : \mathsf{Cmd}^{l_1}(T_2) \qquad P \sim_\gamma P'}{n\,\langle b, o, \sigma, s\rangle \parallel P \sim_\gamma P'} \qquad \frac{\gamma, H \vdash s : \mathsf{Cmd}^{l_1}(T_2) \qquad P \sim_\gamma P'}{P \sim_\gamma n\,\langle b, o, \sigma, s\rangle \parallel P'}$$

Fig. 5. The low-equivalence relation $\sim_\gamma$

From the definition of $\sim_\gamma$ we see that any typable command is equivalent to itself. Two commands are also equivalent if they both only have high side-effects. Commands with only high side-effects are also equivalent to skip-s.

Two local states are equivalent if the values of variables with low types are equal. Two objects are equivalent if the values of fields with low types are equal. The notion of equivalence is then extended to program configurations. We can now define the notion of non-interference we are considering. It is typical for the non-deterministic treatment of information flows, dating back to [24].

Definition 3 (Non-interference). *A program* $\overline{Cl}\ \{\overline{T\ x}\ s; x_0\}$ *is* non-interferent *if for any three states* σ_0*,* $\sigma_0^\bullet$ *and* σ_1 *satisfying* $\sigma_0 \sim_{\overline{x:T}} \sigma_1$*,*

$$b_0[n_0, n_0] \parallel n_0 \langle b_0, \mathsf{null}, \sigma_0, s; \mathsf{release}_L; x_0\rangle \overset{*}{\rightsquigarrow} n_0 \langle b_0, \mathsf{null}, \sigma_0^\bullet, x_0\rangle \parallel \ldots$$

implies that there exists a state $\sigma_1^\bullet$ *with* $\sigma_1^\bullet(x_0) = \sigma_0^\bullet(x_0)$ *and*

$$b_0[n_0, n_0] \parallel n_0 \langle b_0, \mathsf{null}, \sigma_1, s; \mathsf{release}_L; x_0\rangle \overset{*}{\rightsquigarrow} n_0 \langle b_0, \mathsf{null}, \sigma_1^\bullet, x_0\rangle \parallel \ldots \quad .$$

Now we can prove the lemmas and the theorem for non-interference, stating that well-typed programs are non-interferent. Due to space constraints, we will just state the theorems here, and refer to [15] for the proofs and the necessary lemmas.

Theorem 1 (Subject reduction). *If* P_1 *and* P_2 *are well typed under* γ *and* $P_1 \sim_\gamma P_2$ *then if* $P_1 \rightsquigarrow P_1'$ *then there exists* P_2' *such that* $P_2 \rightsquigarrow^* P_2'$ *and* $P_1' \sim_\gamma P_2'$*.*

Theorem 2 (Non-interference). *If* $\vdash Pr : ok$*, where* $Pr = \overline{Cl}\ \{\overline{T\ x}\ s; x_0\}$*, then* Pr *is non-interferent.*

4 Related Work

The treatment of secure information flow in the language- and lattice-based setting is considered to have been pioneered by Denning and Denning. The first well-known type-based analysis for secure information flow in a simple imperative language was proposed by Volpano et al. [25]. Later, their analysis has been extended in many different directions, including treated language constructs, and the versatility of the tools for defining information flow properties. Our analysis, applied to a complex language, draws ideas from the developments in many of those directions. Let us give an overview of those.

While at first, the definitions of secure information flow were given in terms of distinguishable memories, bisimulation relations over program configurations [13, 18] soon emerged as a convenient and composable way of defining information flow properties. The use of weak bisimulations, allowing stuttering, appeared in [23].

Object-oriented features, including fields and methods, were first treated in the JFlow (Jif) compiler [14]. However, they did not provide formal non-interference results. Such results for an OO-language were provided in [3]. In that area, a lot of attention has also been devoted to the analysis of low-level OO-languages, e.g. Java bytecode [4, 5].

Concurrent languages, with possible race conditions and synchronization primitives, bring their own challenges. Secure information flow in a language with the possibility to spawn new threads was first considered in [24]. Synchronization primitives were considered in [19]. A bisimulation-based definition of secure information flow was provided in [7]. In this area, most of the research seems to have concentrated on languages with parallel threads operating on a shared state. For the treatment of processes with private states, one may have to refer to the work based on process calculi [2, 11, 6]. Another interesting area is the building of distributed systems [26] satisfying certain information-flow properties.

In the analysis of thread pools with shared state, the properties of schedulers play a major role in the analysis of information flow properties. Their effect was first considered in [20]. More recently, scheduler strategies for providing the security of information flow have been considered [17, 16].

5 Conclusions

We have demonstrated a type-based information flow analysis for a rich modeling language that has been designed to be applicable in designing large systems. As such, the type-based technique is a suitable choice because of its efficiency in checking large artefacts.

Our work demonstrates that the notion of futures, heavily employed by the language, may cause some interesting information flows in the system. These information flows are particularly apparent if the futures are considered as first-class values. In particular, the synchronization points they create can interfere with the scheduling decisions. Our work shows that the details of scheduling in ABS may need some further design efforts.

Our analysis has been applied to a language employing many different features. Our work has been valuable in pointing out how these features interact with each other in terms of possible information flows. We believe that our work will be helpful in making information flow type systems more widely used in the design and programming phases of the software development process.

References

1. *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*. IEEE Computer Society, 2001.
2. M. Abadi. Secrecy by Typing in Security Protocols. In M. Abadi and T. Ito, editors, *TACS*, volume 1281 of *Lecture Notes in Computer Science*, pages 611–638. Springer, 1997.
3. A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *CSFW*, pages 253–. IEEE Computer Society, 2002.
4. G. Barthe and T. Rezk. Non-interference for a JVM-like language. In J. G. Morrisett and M. Fähndrich, editors, *TLDI*, pages 103–112. ACM, 2005.
5. G. Barthe, T. Rezk, and D. A. Naumann. Deriving an Information Flow Checker and Certifying Compiler for Java. In *IEEE Symposium on Security and Privacy*, pages 230–242. IEEE Computer Society, 2006.

6. C. Bernardeschi, N. De Francesco, and G. Lettieri. Concrete and Abstract Semantics to Check Secure Information Flow in Concurrent Programs. *Fundamenta Informaticae*, 60(1-4):81 – 98, 2004.
7. G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.*, 281(1-2):109–130, 2002.
8. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In R. D. Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007.
9. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
10. R. Hähnle, E. B. Johnsen, B. M. Østvold, J. Schäfer, M. Steffen, and A. B. Torjusen. Report on the Core ABS Language and Methodology: Part A. Highly Adaptable and Trustworthy Software using Formal Models (HATS), Deliverable D1.1A, 4 2010.
11. K. Honda, V. T. Vasconcelos, and N. Yoshida. Secure Information Flow as Typed Process Behaviour. In G. Smolka, editor, *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer, 2000.
12. E. B. Johnsen, J. C. Blanchette, M. Kyas, and O. Owe. Intra-Object versus Inter-Object: Concurrency and Reasoning in Creol. *Electr. Notes Theor. Comput. Sci.*, 243:89–103, 2009.
13. H. Mantel and A. Sabelfeld. A Generic Approach to the Security of Multi-Threaded Programs. In *CSFW* [1], pages 126–.
14. A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL*, pages 228–241, 1999.
15. M. Pettai and P. Laud. Securing the Future — an Information Flow Analysis of a Distributed OO Language. Technical Report T-4-14, Cybernetica AS, 2011.
16. A. Russo, J. Hughes, D. A. Naumann, and A. Sabelfeld. Closing Internal Timing Channels by Transformation. In M. Okada and I. Satoh, editors, *ASIAN*, volume 4435 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2006.
17. A. Russo and A. Sabelfeld. Security for Multithreaded Programs Under Cooperative Scheduling. In I. Virbitskaite and A. Voronkov, editors, *Ershov Memorial Conference*, volume 4378 of *Lecture Notes in Computer Science*, pages 474–480. Springer, 2006.
18. A. Sabelfeld. Confidentiality for Multithreaded Programs via Bisimulation. In M. Broy and A. V. Zamulin, editors, *Ershov Memorial Conference*, volume 2890 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2003.
19. A. Sabelfeld and H. Mantel. Securing Communication in a Concurrent Language. In M. V. Hermenegildo and G. Puebla, editors, *SAS*, volume 2477 of *Lecture Notes in Computer Science*, pages 376–394. Springer, 2002.
20. A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-Threaded Programs. In *CSFW*, pages 200–214, 2000.
21. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In T. D'Hondt, editor, *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer, 2010.
22. G. Smith. A New Type System for Secure Information Flow. In *CSFW* [1], pages 115–125.
23. G. Smith. Probabilistic Noninterference through Weak Probabilistic Bisimulation. In *CSFW*, pages 3–13. IEEE Computer Society, 2003.
24. G. Smith and D. M. Volpano. Secure Information Flow in a Multi-Threaded Imperative Language. In *POPL*, pages 355–364, 1998.

25. D. M. Volpano, C. E. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
26. L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using Replication and Partitioning to Build Secure Distributed Systems. In *IEEE Symposium on Security and Privacy*, pages 236–250. IEEE Computer Society, 2003.