A Secrecy-Preserving Language for Distributed and Object-Oriented Systems☆
March 6, 2018

Toktam Ramezanifarkhani, Olaf Owe, and Shukun Tokas[1]

Department of Informatics, University of Oslo, Oslo, Norway

Abstract

In modern systems it is often necessary to distinguish between confidential (low-level) and non-confidential (high-level) information. Confidential information should be protected and not communicated or shared with low-level users. The *non-interference* policy is an information flow policy stipulating that low-level viewers should not be able to observe a difference between any two executions with the same low-level inputs. Only high-level viewers may observe confidential output. This is a non-trivial challenge when considering modern distributed systems involving concurrency and communication.

The present paper addresses this challenge, by choosing language mechanisms that are both useful for programming of distributed systems and allow modular system analysis. We consider a general concurrency model for distributed systems, based on concurrent objects communicating by asynchronous methods. This model is suitable for modeling of modern service-oriented systems, and gives rise to efficient interaction avoiding active waiting and low-level synchronization primitives such as explicit signaling and lock operations. This concurrency model has a simple semantics and allows us to focus on information flow at a high level of abstraction, and allows realistic analysis by avoiding unnecessary restrictions on information flow between confidential and non-confidential data.

Due to the non-deterministic nature of concurrent and distributed systems, we define a notion of *interaction non-interference* policy tailored to this setting. We provide two kinds of static analysis: a secrecy-type system and a trace analysis system, to capture inter-object and network level communication, respectively. We prove that interaction non-interference is satisfied by the combination of these analysis techniques. Thus any deviation from the policy caused by implicit information leakage visible through observation of network communication patterns, can be detected. The contribution of the paper lies in the definition of the notion of *interaction non-interference*, and in the formalization of a secrecy type system and a static trace analysis that together ensure interaction non-interference. We also provide several versions of a main example (a news subscription service) to demonstrate network leakage.

Keywords: Concurrent Objects; Asynchronous Methods; Communication Patterns; Non-interference; Information Flow; Secrecy; Confidentiality; Distributed Systems; Network Leakage; Inter-Object Leakage.

1. Introduction

Programming languages can provide fine-grained control for security issues because they allow accurate and flexible security information analysis of program components [1]. In particular, to specify and enforce information-flow policies, the effectiveness of language-based techniques has been established. Information-flow policies are essentially specified based on a mapping from the set of logical information holders to a lattice of security classes representing levels of information sensitivity. Moreover, these policies usually

[1]email: {toktamr,olaf,shukunt}@ifi.uio.no

dictate that no execution of the program should lead to an information-flow from more sensitive to less sensitive information holders [2], otherwise the information-flow is called "illegal".

Since information-flow policies are hyper-properties [3], i.e., are characterized as sets of trace properties, their specification, enforcement, and reasoning are difficult, especially in complicated systems. Therefore, giving a precise definition of the policy regarding legal and illegal information flows is challenging and highly dependent on the model of the system, the attackers, and their capabilities. Secure information flows are often expressed by semantic models of program execution in the form of a *non-interference* policy. Non-Interference stipulates that manipulation and modification of confidential data should be allowed in programs, as long as their visible outputs do not improperly reveal information about the confidential data. In addition, attackers are typically assumed to be able to view "low" information. The usual method for showing that non-interference holds is to demonstrate that the attacker cannot observe any difference between two executions that differ only in their confidential input [4]. In other words, if two possible input states of a program share the same low values, then the observable behaviors of the program execution on these states should be indistinguishable by the attacker [5]. Although the observable behavior is defined by the program output, there is no limitation in specifying program behaviors, and there is no fixed limitation on what is observable by the attackers. For example, when programs have runtime interactions with the environment, attackers may also see intermediate outputs [6]. In addition, the attacker may observe the progress of the program, e.g., absence or presence of the next observable value, which leads to the concept of progress-sensitive non-interference [6].

In the setting of distributed concurrent objects communicating by asynchronous methods calls, variables are encapsulated by objects and are not directly observable when forbidding remote variable access. Thus illegal explicit flows in the sense of assignment of confidential (or high) variables to non-confidential (or low) variables inside objects are not critical. In this setting, method calls and replies are represented by messages sent over a network, and thus network traffic could be observable to attackers. Therefore, patterns of network messages reflecting calls and replies can be informative to attackers and may reveal high-level information. For example, consider the following code in which for a specific user role the program's privileges are temporary raised to allow the creation of a new user folder[2]:

```
try:
        if (URole):
                o.raisePrivileges()
                os.mkdir('/home/' + username)
                o.lowerPrivileges()
except OSError:
        print('Unable to create new user directory')
        return False
return True
```

where the syntax $o.m(e)$ denotes a remote method call to o. An attacker may deduce confidential information about the user role based on the observation of method calls because in the then-branch there is a call and in the else-branch, which is empty in this case, there is no method call. This is a case of implicit information flow, which may appear when the observable program behavior includes observation of method calls. In addition, such information leakage can result in other successful critical attacks such as arbitrary code execution in injection attacks, which have been among the Top Ten critical attacks for years [7]. Here the essential information that makes the attack successful, is related to when and how to attack the program. For instance in the above example, the attacker knows that in some executions the program only raises the privilege level for a short while before lowering it again. After observation of the call in an execution, he can find a useful step toward a successful attack, for instance by throwing an exception, which leads away from the call to *lowerPrivileges*(). As a result, the program is indefinitely operating in a raised privilege

[2]This vulnerability has been exploited in SplitVT, which is a program for splitting terminals into two shells, and allowing arbitrary code execution by attackers CVE-2008-0162.

state, possibly allowing further exploitation to occur by the attacker such as calling privileged functions [8], executing the attacker's own code and launching an injection attack.

To prevent information leakages in distributed concurrent object systems, we enrich the notation of non-interference by considering observability of interactions among objects by attackers. So, we introduce a notion of *Interaction Non-Interference*, which stipulates program executions to be equivalent in the view of attackers observing method call events. In addition, we are considering prevention of *inter-object leakages* when an object improperly sends secret information to another object, that might be non-observable from the network view. We also consider some special cases of network leakage based on sophisticated mechanisms including suspension and non-blocking calls, which increase difficulties in specification and enforcement of non-interference. For example, attackers might be able to distinguish between executions with the same sequence of method calls that only differ in blocking or suspension behavior. However, we do not impose unnecessary restrictions on information flows from more sensitive to less sensitive variables inside objects, which make our approach more realistic than other pessimistic approaches based on static analysis, and thus significantly reduces the rate of false positives.

Our setting. To formalize our approach we consider a high-level core language based on the chosen concurrency model, namely the paradigm of so-called *active objects* using asynchronous method calls as the only interaction mechanism, thereby combining the Actor model and object-orientation. This language is derived from *Creol* [9]. Shared variables as well as thread-based notification are avoided. Synchronization control is achieved by cooperative scheduling: A local suspension mechanism allows an object to perform other tasks while waiting for a condition to become true or for a method result to appear. In Creol, an object can be seen as a black box in the sense that its content such as its fields are not observable from outside the object, and the main observation of objects is through their interaction by means of method calls. In a network this is observed through messages corresponding to invocations and completions of remote method calls, and dynamic object creation. Underlying network protocols may ensure that an attacker may observe but not alter the content of a message [2], and message content may be considered non-observable due to encryption techniques [10]. We consider the case that the destination and source of a message is considered observable, and in addition, we assume that an observer may be able to deduce the method name and whether it reflects a method invocation or a method completion. In addition, confidential message content communicated through the network to untrusted objects is also a source of secrecy leakage. Our approach covers these kinds of information leakages, and is relevant in Actor-based systems since these are based on message interaction. The notion of observable events may be further refined, for instance by considering certain parts of the network secure, say locally created objects.

We present an extension of Creol called *SeCreol*, in which Creol is extended with awareness of secrecy levels as well as secrecy type information. We show that programs respecting certain static restrictions, including secrecy typing, satisfy interaction non-interference. Moreover, to ensure that a system preserves secrecy of information, we use a combination of access control and information flow control of communicated information capturing direct access, and tracking communication patterns to capture indirect accesses.

Contribution. The following are the main contributions of this paper: (i) Introducing interaction non-interference for non-deterministic distributed systems communicating by message passing. (ii) Extending the core Creol language by providing a security-type system and developing a static analysis approach for detection of network leakage. (iii) Provably enforce the interaction non-interference property in programs of the SeCreol language by static analysis.

There is evidence, for example in [8], showing that the interaction between components and objects, e.g., the sequence of system and method calls, are valuable for attackers and can cause information leakage. Therefore, interaction non-interference is a critical property in a variety of secure systems, and thus its satisfaction and application are not limited to object-oriented systems.

Paper outline. Section 2 formalizes the observable behavior of object-oriented distributed systems by explaining their execution model, and discusses security and attack models, as well as system assumptions, leading to the notion of interaction non-interference in Section 3. Section 4 introduces the SeCreol core language and a subscription example, to demonstrate our approach. A type system for secrecy levels is given in Section 5, while network leakage is considered in Section 6. Section 7 shows soundness of the network analysis, using the operational semantics of SeCreol given in Appendix A. Section 8 discusses related work, and Section 9 concludes the paper points, and suggests possible future work.

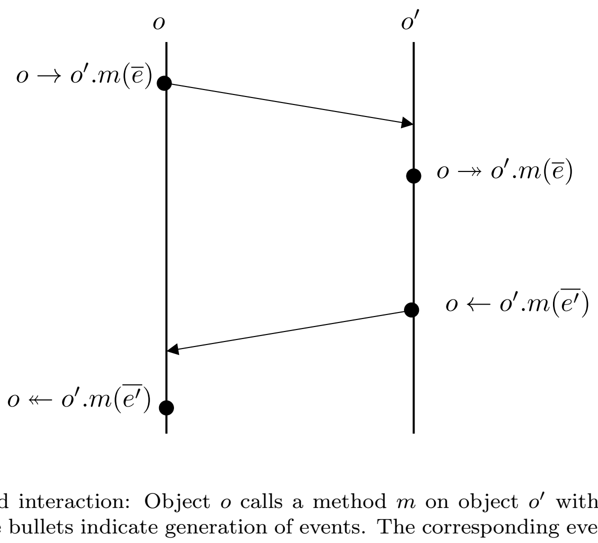
2. Behavior of Object-Oriented Distributed Systems

We consider concurrent, distributed objects where each object has its own execution thread. An object does not have access to the internal state variables of other objects. Object communication is only by method calls, allowing asynchronous communication, implemented by means of asynchronous message passing. In order to avoid undesirable waiting in the distributed setting, we allow mechanisms for non-blocking method calls. By means of a suspension mechanism, unfinished method invocations in an object may be placed on the object's process queue, for instance while waiting for a response from another object. The process will be enabled when the object receives the response. This allows flexible interleaving of incoming calls and (enabled) suspended processes. Internally in an object, there is at most one process executing at any time. Objects reflect concurrent system components, while data structure inside an object is defined by data types using functional programming.

The execution of a distributed system can be represented by the sequence of communication events that has appeared between the system components. This sequence is called the *communication history* (or *trace*) [11, 12, 13], which in our case consists of invocation and completion events of the called methods. At any point in time, the communication history abstractly captures the system state [14]. And we represent the set of executions of a distributed system by its possible communication histories, letting infinite histories represent non-terminating executions. The formalization of interaction non-interference, which we define later, is given as a property over the communication history.

Definition 1. ***(Communication events)*** *We consider the following events* Ev *of a system, where* o, o' *are objects,* m *is a method, and* $\overline{e}$ *is a list of expressions:*

- *the set of invocation events* $o \rightarrow o'.m(\overline{e})$*,*
- *the set of invocation reaction events* $o \twoheadrightarrow o'.m(\overline{e})$*,*
- *the set of completion events* $o \leftarrow o'.m(\overline{e})$*,*
- *the set of completion reaction events* $o \twoheadleftarrow o'.m(\overline{e})$*,*
- *the set of object creation events* $o \leftrightarrow o'.\textbf{new}\,C(\overline{e})$



o
o'
$o \rightarrow o'.m(\overline{e})$
$o \twoheadrightarrow o'.m(\overline{e})$
$o \leftarrow o'.m(\overline{e'})$
$o \twoheadleftarrow o'.m(\overline{e'})$

Figure 1: Illustration of method interaction: Object o calls a method m on object o' with arguments $\overline{e}$. The long arrows indicate message passing, and the bullets indicate generation of events. The corresponding event is written next to each bullet.

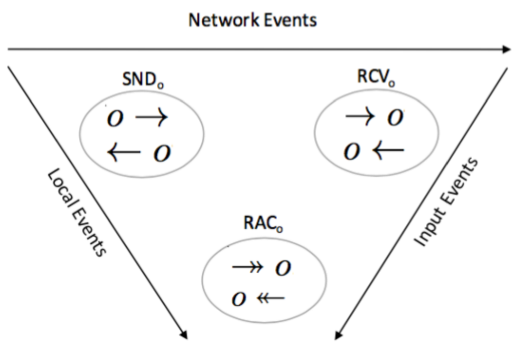


Figure 2: Illustration of the categories of method-related events for an object o. Observable network events for o consist of events from o (SND$_o$) and events to o (RCV$_o$), while non-observable o events are the internal reaction events of o (RAC$_o$).

The arrows reflect the direction of the message sending, and a two-way arrow indicates a synchronization event.

In our model, a method call to m is reflected by the four communication events

$$o \rightarrow o'.m(\overline{e}); o \twoheadrightarrow o'.m(\overline{e}); o \leftarrow o'.m(\overline{e}'); o \twoheadleftarrow o'.m(\overline{e}')$$

as graphically illustrated in Fig. 1. The figure shows the time-line of the caller object o and the callee object o'. An invocation message is sent from o to o' when a method m is called, which is reflected by the invocation event $o \rightarrow o'.m(\overline{e})$ where $\overline{e}$ is the list of actual parameters. The invocation reaction event $o \twoheadrightarrow o'.m(\overline{e})$ reflects that o' starts execution of the method, and the completion event $o \leftarrow o'.m(\overline{e}')$ reflects method termination, where $\overline{e}'$ is the list of returned values. Reading the reply in object o is reflected by the completion reaction event $o \twoheadleftarrow o'.m(\overline{e}')$. Other events may be interleaved with these four events, and in particular there might be an arbitrary delay between message receiving and reaction (due to message queuing).

Object creation is similar to method interaction. The event $o \leftrightarrow o'.\mathbf{new}\, C(\overline{e})$ can be understood as the sequence $o \rightarrow o'.C(\overline{e}); o \twoheadrightarrow o'.C(\overline{e}); o \leftarrow o'.C(void); o \twoheadleftarrow o'.C(void)$ where C represents the class constructor.

Fig. 2 categorizes communication events between objects. Messages sent from an object o are denoted by SND_o and messages received by an object o are denoted by RCV_o, while RAC_o denotes reactions events of the object o, which are internal o events. For a given object o, these three event sets are disjoint.

$$\begin{array}{lll} SND_o & \equiv \{o \rightarrow, \leftarrow o\} & \text{send events of o} \\ RCV_o & \equiv \{\rightarrow o, o \leftarrow\} & \text{receive events of o} \\ RAC_o & \equiv \{\twoheadrightarrow o, o \twoheadleftarrow\} & \text{reaction events of o} \end{array}$$

where $o \rightarrow$ denotes the set of invocation events *from* o and $\leftarrow o$ the set of completion events *from* o, while $\rightarrow o$ denotes the set of invocation events *to* o and $o \leftarrow$ the set of completions events *to* o, and so on. The set SND_o represents output from o, while RCV_o and RAC_o represent (external and internal) input to o. The union of SND_o and RCV_o represents events visible over the network, while RAC_o represents internal events not visible over the network, as illustrated in Fig. 2.

Next, we define *communication histories* as a sequence of events.

Definition 2. ***(Communication histories)*** *The communication history for a system at a given point in an execution is a finite sequence of* Ev *events.*

In our static analysis, we will consider finite traces (histories), representing executions up to a given program point, or segments of such executions. The empty sequence is denoted *empty* and we let semicolon denote sequence append (adding an event to the end of a sequence). Thus any sequence is either empty or can be seen as an appended sequence. We let t/S denote the projection of trace t by a set of events S defined by $\mathit{empty}/S = \mathit{empty}$ and $(t;x)/S = \texttt{if } x \in S \texttt{ then } (t/S);x \texttt{ else } t/S$, and we overload the notation by letting t/o denote the projection of a trace t by the set of events that has the object o as sender or receiver, i.e., the subsequence of events that involve o.

2.1. Attack Model

We consider two levels of attacks: i) *Inter-object leakage*, where attackers appear as objects and improperly obtain secret information from other objects. ii) *Network leakage*, where attackers derive secret information by observing the network traffic.

We consider the case that network attackers may know the whole system including program code and the distribution, but can only observe observable runtime events at the network level. Based on the explained behavioral model of object-oriented distributed systems, these events are passed as messages between objects. Therefore, even when assuming encryption of message content, the source and the destination object of messages are (implicitly) visible for attackers. Knowledge of the program code may allow an attacker to sometimes deduce the methods name in an event and whether it is an invocation or a completion message. By overestimating the attackers' capabilities, we assume that the method name might be known to an attacker, and therefore we assume that all aspects of a message except the parameters are observable. However, reaction events are not observable by attackers since they are internal to an object.

Hence, possible leakage of information includes *network leakages* and *inter-object leakages.* Leakage of network traffic is caused by observing the patterns of network traffic, while leakage to other objects occurs when an object improperly sends secret information to the other object, something that might not be observable from the network view, but from the other object.

Self-calls will not be observable over the network and do not cause network leakage. Similarly, communication to and from internal objects (for instance object generated by this at the same location) could also be considered non-observable at the network level (as well as local communication over internal sub-nets, assumed to be safe). However, for simplicity, we do not here include location awareness and treat all objects as if they were in different locations. (Location awareness could easily be added.)

In a distributed object system, the relative execution speed of the objects is not known. This is reflected in our model by letting the queue of incoming messages to an object be unspecified. Two messages sent from one object to another may be handled in the reverse order. In general, the ordering of RAC_o events in an execution may differ from the ordering of RCV_o events in the same execution. This gives a certain degree of non-determinism at the object level. At the network level one might consider message overtaking, loss, and duplication. However this gives an even higher degree of non-determinism, and therefore a weaker notion of leakage. We therefore ignore message overtaking, message loss, and message resubmission at the network level, since sequence information represents the upper limit of what is observable at the network level, assuming reasonable reliable and efficient networks. The order of messages in a sequence is then observable and can be informative for attackers, and may cause network leakage.

This notion of network observability can be formalized as an equivalence relation over histories, called observable network equivalence ($\approx_{net}$), and will be used to compare the behaviors of two different execution histories (σ and σ') in order to detect leakage.

Definition 3. ***(Observable network equivalence)***

$$\sigma \approx_{net} \sigma' \equiv (\mathit{obs}(\sigma) = \mathit{obs}(\sigma'))$$

where obs expresses observable trace information*, defined inductively over finite histories σ by:*

$$\begin{array}{lcll}
\mathit{obs}(\mathit{empty}) & = & \mathit{empty} & \\
\mathit{obs}(\sigma; (o \rightarrow o'.m(\overline{e}))) & = & \mathit{obs}(\sigma); (o \rightarrow o'.m) & \texttt{if } o \neq o' \\
\mathit{obs}(\sigma; (o' \leftarrow o.m(\overline{e}))) & = & \mathit{obs}(\sigma); (o' \leftarrow o.m) & \texttt{if } o \neq o' \\
\mathit{obs}(\sigma; (o \leftrightarrow o'.\texttt{new}C(\overline{e}))) & = & \mathit{obs}(\sigma); (o \rightarrow o'.C)(o \leftarrow o'.C) & \\
\mathit{obs}(\sigma; x) & = & \mathit{obs}(\sigma) & \texttt{otherwise}
\end{array}$$

The *otherwise* case is taken when no other equation applies (in this case when x is a reaction event). Similarly, observable network equivalence *relative to a particular object* o is defined by

$$obs(\sigma/o) = obs(\sigma'/o)$$

The latter will be used when we do class-wise analysis, focusing on the object represented by this.

3. Interaction Non-Interference

As mentioned, non-interference ensures that an attacker should not be able to obtain confidential information by observing the low input and output of an executions of a system. We therefore need to capture the possible observations at a given point in an execution, represented by the communication history at that point, and define a notion of low equivalence between histories. Intuitively, two histories are low equivalent if they have the same low information, i.e., when ignoring non-observable events and arguments that are not low.

Definition 4. ***(Low equality)*** *$\sigma =_L \sigma'$ is defined by $\mathsf{low}(\sigma) = \mathsf{low}(\sigma')$, defining the low projection over histories and expressions as follows*

$$\begin{array}{lcll}
\mathsf{low}(empty) & = & empty & \\
\mathsf{low}(\sigma; o \rightarrow o'.m(\overline{e})) & = & \mathsf{low}(\sigma); o \rightarrow o'.m(\mathsf{low}_m(\overline{e})) & \\
\mathsf{low}(\sigma; o \leftarrow o'.m(\overline{e})) & = & \mathsf{low}(\sigma); o \leftarrow o'.m(\mathsf{low}_m(\overline{e})) & \\
\mathsf{low}(\sigma; o \leftrightarrow o'.\mathbf{new}\, C(\overline{e})) & = & \mathsf{low}(\sigma); o \rightarrow o'.C(\mathsf{low}_C(\overline{e})); o \leftarrow o'.C(\mathsf{void}) & \\
\mathsf{low}(\sigma; x) & = & \mathsf{low}(\sigma) & \textbf{otherwise}
\end{array}$$

where $\mathsf{low}_m(\overline{e})$ is defined by the sublist of those parameters e_i for which the ith parameter is declared as Low *according to the method declaration. Similarly, $\mathsf{low}_C(\overline{e})$ is the sublist of actual class parameters e_i for which the ith class parameter is declared as* Low.

Observation. It follows directly from the definitions above that low equality is a stronger relation than observable equality, i.e. $\sigma =_L \sigma'$ implies $\sigma \approx_{net} \sigma'$.

Let Σ be the set of (finite or infinite) traces of events for all possible completed executions of a system, letting finite traces represent terminating executions. Below σ and σ' will range over Σ. Non-interference of an object o expresses that if two executions involving o are low equal up to a certain time i, then also the low output will be the same, including the next step (after the given time), if it is an output (i.e., a SND_o event). If the object is deterministic with respect to its input, this could be expressed by

$$\forall \sigma, \sigma', i\,.\ (\sigma/o)|i =_L (\sigma'/o)|i \wedge (\sigma/o)[i+1] \in SND_o \Rightarrow\ (\sigma/o)|\,i+1 \approx_{net} (\sigma'/o)|\,i+1$$

where i represents the time relative to o (the number of o steps), $\sigma[i]$ denotes the ith element of σ ($i \in Nat$), and $\sigma|i$ is the sequence prefix $\sigma[1..i]$ (or σ if the length of σ is less than i).

Since our objects are non-deterministic, due to non-deterministic queues of incoming messages and of internal process queues, which in turn reflect non-deterministic network speed and object processing speeds, this definition cannot be used. It would be too strong, since it essentially expresses that the next low output (if any) from a given object o at a given time is deterministic. For instance if one possible execution at a given time starts with a low output event x from o and another with an observally different low output event x' from o, interference would not be satisfied, since the execution that starts with x has next a low output that is not observably the same as in all other executions.

In order to deal with non-determinism, we need to consider the set of possible executions, such that observable network equivalence holds for the set of possible next steps, i.e., considering all possible continuations of σ after i compared to the *set of* all possible continuations of σ' after i. In general, equivalence of two sets can be expressed by using existential quantification. We therefore express our notion of non-interference using an existential quantifier, as follows.

Pr	$::=$	$In^{*}\ Cl^{*}$	program
$\mathcal{L}$	$::=$	$\mathsf{Low} \mid \mathsf{High} \mid ...$	secrecy levels
T	$::=$	$I \mid \mathsf{Int} \mid \mathsf{Any} \mid \mathsf{Bool} \mid \mathsf{String} \mid \mathsf{Void} \mid \mathsf{List}[T] \mid ...$	types, here [and] are ground symbols
U	$::=$	$T \mid T : \mathcal{L}$	type and secrecy level (default is Low)
In	$::=$	$\mathbf{interface}\ I\ [\mathbf{extends}\ I^{+}]\ [\mathbf{with}\ I]\{D^{*}\}$	interface declaration
Cl	$::=$	$\mathbf{class}\ C\ ([U\ cp]^{*})\ [\mathbf{implements}\ I^{+}]$	
		$\{[U\ w\ [:= e]]^{*}\ [B]\ [[\mathbf{with}\ I]\ M]^{*}\}$	class definition
M	$::=$	$D\ B$	method definition
D	$::=$	$U\ m([U\ y]^{*})$	method signature
B	$::=$	$\{[T\ x\ [:= e];]^{*}\ [s;]\ \mathbf{return}\ e\}$	method blocks
v	$::=$	$x \mid y \mid w$	variables (local or field)
e	$::=$	$\mathsf{null} \mid \mathsf{void} \mid \mathsf{this} \mid \mathsf{caller} \mid cp \mid v \mid f(\overline{e}) \mid e \sqsubseteq e$	pure expressions, including level-checking
rhs	$::=$	$e \mid \mathbf{new}\ C(\overline{e})[:\mathcal{L}] \mid e.m(\overline{e})$	right-hand-sides
s	$::=$	$\mathsf{skip} \mid s;s \mid v := rhs$	assignment
		$\mid e!m(\overline{e}) \mid \mathbf{await}\ v := e.m(\overline{e}) \mid \mathbf{await}\ e$	call statements and suspension
		$\mid \mathbf{if}\ e\ \mathbf{then}\ s\ [\mathbf{else}\ s]\ \mathbf{fi} \mid \mathbf{while}\ e\ \mathbf{do}\ s\ \mathbf{od}$	if and while statements

Figure 3: SeCreol BNF language syntax, with C denoting class name, I interface name, m method name, cp formal class parameter, w fields, y method parameter, x local variable. We let $[\ ]^{*}$, $[\ ]^{+}$ and $[\ ]$ denote repeated, repeated at least once, and optional parts, respectively.

Definition 5. ***(Interaction non-interference)*** *We define* interaction non-interference *(INI_o) for an object (or a group of objects) o:*

$$\forall \sigma, \sigma', i \,.\, (\sigma/o)|i =_L (\sigma'/o)|i \wedge (\sigma/o)[i+1] \in SND_o \Rightarrow\ \exists \sigma'' \,.\, (\sigma'/o)|i \leq (\sigma''/o) \wedge (\sigma/o)|i+1 \approx_{net} (\sigma''/o)|i+1$$

where $\leq$ expresses the sequence prefix relation.

Here $\sigma, \sigma', \sigma''$ range over sequences of events reflecting possible completed executions (Σ). Thus the definition implies liveness (progress sensitivity). Class-wise analysis implies that we are interested in a given object o. The existential quantifier reflects possible non-determinism, and σ'' allows to choose the non-deterministic extension of σ' that follows σ for the given object o. The definition says that if an execution ($sigma$) has an output from o at time $i+1$ then any other execution with the same low o events up to time i has a possible extension (σ'') after time i with the observably same output event. Thus the set of observable output events for a given object at a given time must be deterministic relative to the low inputs before this time.

This definition of interaction non-interference is sufficient to avoid leakage by network attackers.

Our goal is to statically detect the INI_o property by means of two kinds of static analysis: i) a deductive system for secrecy typing ii) a deductive system for trace analysis of network events, such that both analyses are class-wise. In order to show this in some detail we will consider a high-level core language for the chosen concurrency model.

4. The SeCreol Language

We define a minimal high-level language illustrating the concurrency model of concurrent objects communicating with asynchronous methods. The language, called SeCreol, builds on the concurrency model of Creol [9], extended with secrecy constructs, including declaration of static secrecy levels for variables and parameters, and testing of runtime secrecy levels of objects.

The syntax of SeCreol is given in Fig. 3. A program consists of a number of interfaces and classes. We let the last class declared in the program be taken as the main class, which is instantiated automatically and its body will start to execute. An interface may have a number of super-interfaces and method declarations.

A method of an interface or class may have a *cointerface*, which gives the (minimal) interface of the caller objects. (For simplicity an interface may only use one common cointerface for all its methods.) This allows type-correct call-backs [9]. A class C takes a list of class parameters cp, defines fields w, and has an optional method body for initialization B (also called the class constructor), followed by method definitions, with the corresponding cointerfaces as declared in the interfaces. Class parameters cp are like fields apart from being initialized through the **new** statement. Class parameters, the implicit class parameter this and the implicit method parameter caller are read-only. A class may implement a number of interfaces, and for each method of an interface of the class it is required that the class defines the method such that the cointerface and types of each method parameter and return value are respected. Additional methods may be defined in a class as well, but these may not be called from outside the class. For simplicity we omit class inheritance.

All variables and parameters are typed by data types or interfaces and for simplicity the syntax of the data type language is omitted here. Classes are not allowed as types, which means that an object can only be seen though an interface, and therefore, remote access to fields nor methods that are not exported through an interface is not allowed. This limits the possible interactions between the concurrent objects, regardless of where they are located, and in particular, shared variables concurrency is avoided. With respect to security analysis, it has the advantage that no field is observable from outside of an object. Thus observable behavior is limited to interaction by means of method-oriented communication.

Expressions e and functions f are side-effect free, and $\overline{e}$ is a (possibly empty) expression list, comma-separated. Statements include standard constructs for assignment, skip, if, while, object generation, and sequential composition. The *simple call* statement $e!m(\overline{e})$ is like message passing; a message is sent to the object expressed by e (the callee) indicating that it should execute method m (when the callee is free to do this) with a list of actual parameters $\overline{e}$. Thus the current object is not blocked, and will not receive the return value. If the return value is desired by the calling object, it may use the *blocking call* statement $v := e.m(\overline{e})$ or the *non-blocking call* statement **await** $v := e.m(\overline{e})$. The latter call statement forces the caller object to suspend the current process, allowing it to continue with any enabled suspended process in its process queue or perform an incoming call. Similarly, the conditional await statement **await** e suspends, placing the current process on the process queue. This process is enabled when the Boolean condition e is satisfied. The considered core language allows high-level and yet efficient method-based interaction between concurrent objects, supporting both passive and active waiting. The operational semantics of the language is given in Appendix A.

The language is strongly typed, and a typing system can be given in the style of [15]. We use a standard notion of subtyping (subsuming subinterfacing). If T' is a subtype/subinterface of T, we say that T' is better than T, and a method declaration D' is better than D if they have the same method name and number of parameters, the return type of D' is better than that of D, and each formal parameter of D is better than the corresponding one of D' (i.e., contravariance). The type system will ensure that a class properly defines all the methods of its declared interfaces (and superinterfaces), or better ones, and it ensures that each method call will be bound to a method declaration. The self-call $\mathsf{this}.m(\overline{e})$ will be bound to the enclosing class (which must have a type-correct declaration of m). When e is of interface I, the method call $e.m(\overline{e})$ will be bound to I (or the closest superinterface of I with a (type-correct) declaration of m). For simplicity, we do not allow interfaces nor classes to declare several methods with the same name. Interface Any is the most general interface, supported by any object.

A variable is typed either by an interface or by a data type, called *object variable* or *data variable*, respectively. The runtime value of an object variable is an object identity (or null), and that of a data variable is a data value. Data variables are passed by value and object variables are passed by reference (i.e., the object identity is passed by value). Note that all object expressions are typed by an interface, except this, which is typed by the enclosing class. In a well-typed program, we may assume that each call is annotated by the interface/class of the callee, as in $o.m_I(\overline{e})$ where I will contain a declaration of m.

Secrecy Levels. We enrich the typing system with *secrecy levels*. Secrecy levels range over $\mathcal{L}$ of basic secrecy descriptions with ordering $\sqsubseteq$, such that $(\mathcal{L}, \sqsubseteq)$ is a lattice, i.e., a partially ordered set with meets ($\sqcap$), joins ($\sqcup$), a top element $\top$ and bottom element $\bot$. Higher in the lattice means more secure; and thus the top element is the most secure. For example, a simple multi-level secrecy system might have secrecy descriptions *low*, *medium*, and *high*, with ordering $low \sqsubseteq medium \sqsubseteq high$, where $low = \bot$ and $high = \top$. A more expressive lattice could have several medium elements, indexed by object identities, or sets of object identities, for controlling access rights. This is essential at runtime for controlling secrecy with respect to objects; however, in our static analysis we will not use levels indexed by identities, since in general there is limited static knowledge about object identities.

In the syntax, all fields, formal parameters, and return values are given a secrecy level, with level *low* as default (if none is specified). Local variables do not have a declared secrecy level; their level starts as Low but may change after each statement. At runtime, objects are assigned a secrecy level that protects against unauthorized changes. Such a protected part is typical in policy enforcement research [16]. The statically assigned level of a formal data parameter represents the maximal level of any actual parameter. The declared secrecy level of an object variable expresses the secrecy of the object identity, which is typically *low*, reflecting that object *identities* (as such) are considered non-secret, whereas the runtime secrecy level of an object gives more detailed information, for instance about the access rights of the object. The static analysis is class-based, and therefore the analysis is based on the (statically) declared levels, and not the runtime object levels. However, the language allows specification of restrictions on the secrecy level of a new object (as in x:=new C():Low) which determines the initial runtime secrecy level of the generated object. At runtime an object generated by the statement x:=new C():l will get the level $l \sqcap l_{\mathsf{this}}$ where l_{this} is the level of the parent object. Note that $l \sqcap l_{\mathsf{this}} \sqsubseteq l_{\mathsf{this}}$, which ensures that the secrecy level of the generated object will not exceed that of the parent object. As an object encapsulates local data and fields, these are not accessible from outside of the object, and we do not need static control of write access to fields of an object. At runtime the secrecy level of an object can be tested using the $\sqsubseteq$ operation in the program.

In the static analysis, we consider all statically assigned levels, and all possibilities for levels that can be assigned at runtime. This allows us to detect a maximal secrecy level for each program variables at any given point in a program (see Sec. 5). The next subsection describes an example of a network leakage, as well as a non-leaking version.

A Subscription Example

A simple subscription example illustrates the different language mechanisms, including simple, blocking and non-blocking method calls, and suspension mechanisms. Note the use of cointerfaces in Fig. 4, which implies that the caller of subscr is of type Client, which in turn allows the field *users* in class SUBSCLIENT to be typed as a list of Clients, thereby allowing the call users[i]!notify(n) in the class to be type correct since users[i] then is of interface type Client. Here List is a predefined generic data type with generators empty and insert, and with functions length and delete. All program variables in the example are declared as Low (by default) except the parameters to notify and publish, allowing high level News information to be passed to clients through these methods. The data type News may be defined as a String or a more complex data structure. To control and limit the notification of high level news, the test $n \sqsubseteq first(user)$ is made before notification. Thereby notification is restricted to client objects with a high enough runtime secrecy level.

We use the convention that class names are written in upper-case, interfaces and types are capitalized, while variable, method, and function names are in lower-case characters. We omit return void at the end of a class body. The non-blocking call await v := c. notify (n) makes a NEWSPROVIDER object send notifications at a speed adjusted to the Client c, without blocking itself from responding to other calls. In contrast, the *notify* method in class SUBSCLIENT uses a simple call, users [i]! notify (n), to notify each client, without suspending nor waiting for each one to receive the call. Also, the main program (i.e., the constructor of the main class) uses simple calls, in order to set up the initial system structure without waiting for replies. The suspending call in make_subscr allows the SUBSCLIENT object to continue with notifications and other requests while waiting for Subscriber s to respond. The main program declares a subscriber $s1$ getting news from $n1$ and $n2$, and $s1$ notifies c and $s2$, while $s2$ further notifies a and b, as illustrated in Fig. 5. Note that $s1$ and $s2$ play a dual role, that of a client and that of a subscriber, and must therefore be of interface SubsClient. This makes the program well-typed.

The first version of the example poses a possible network leakage because a network observer may detect which subscribed client objects are High, by comparing the network traffic from a given subscriber object over time. The second version uses a dummy call in the else branch of notify to confuse a network observer

```
interface Client {
        Void notify(News:High n)
        Bool make_subscr(s:Subscriber) }
interface Subscriber with Client {
        Bool subscr()
        Bool unsubscr() }
interface SubsClient extends Subscriber, Client {}
interface Publisher {Void publish(News:High n)}
class SUBSCLIENT() implements SubsClient{
        List[Client] users := empty; // to store subscribers
        Void notify(News:High n) {Nat i:=1;
                while (i ≤ length(users)) do
                  if n ⊑ users[i] // checking runtime sec.levels
                  then users[i]!notify(n) fi; // simple call
                  i:=i+1 od;
                return void}
        Bool make_subscr(s:Subscriber){Bool ok;
                await ok:= s.subscr(); return ok}
     with Client
        Bool subscr(){
                if caller ≠ this
                then users := insert(users,caller) fi;
                return caller ≠ this }
        Bool unsubscr(){
                users := delete(users,caller);
                return true} }
class NEWSPROVIDER(Client c) implements Publisher {Void v;
            Void publish(News:High n){v := await c.notify(n); return v}}
class MAIN() { {// main program
                SubsClient s1 := new SUBSCLIENT():High;
                SubsClient s2 := new SUBSCLIENT():High;
                Client a := new SUBSCLIENT();
                Client b := new SUBSCLIENT():High;
                Client c := new SUBSCLIENT();
                Publisher n1 := new NEWSPROVIDER(s1):High;
                Publisher n2 := new NEWSPROVIDER(s1);
                s2!make_subscr(s1);
                a!make_subscr(s2);
                b!make_subscr(s2);
                c!make_subscr(s1) } }
```

Figure 4: A simple subscription example.

(which cannot see the news content). The third version reduces the need for dummy calls due to non-deterministic background (self) activity that makes dummy calls when the object is not busy with other things. Here a somewhat different version of *notify* is given.

It is noticeable that in our approach we are considering the worst case scenario. For instance, in the subscription example, we consider that the set of subscribed and unsubscribed clients are known to the attacker (i.e., by tracking the execution communication), otherwise an observer may not be sure that the lack of notification to a client is due to unsubscription or the presence of high-level news.

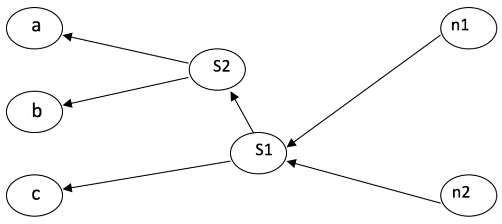


Figure 5: Illustration of the flow of news for the subscribers, clients, and news providers in the main program.

```
class SUBSCLIENT() implements SubsClient {
        ... // as before

        Void notify(News:High n) {Nat i:=1;
                while (i ≤ length(users)) do
                  if n ⊑ users[i] // checking runtime sec.levels
                  then users[i]!notify(n)
                  else users[i]!notify(empty) fi; // async. call
                  i:=i+1 od;
                return void }
}
```

Figure 6: Secure Class Server, second version – more secure, but generates dummy calls.

```
class SUBSCLIENT() implements SubsClient { List[Client] users := empty;
        { this!mask() } //initialization, starting internal background
activity by a self-call
        ...
        Void notify(News:High n) { Nat i:=1;
            if improper(n) then skip
            else while (i ≤ length(users)) do
                users[i]!notify(n);
                i:=i+1 od fi;
                return void}
        // send to all or none

        Void mask(){Nat i:=1;
                while (i ≤ length(users)) do
                  await true;
                  users[i].notify(empty) ; // suspending call
                  i:=i+1 od;
                this!mask();
                return void}
}
```

Figure 7: Secure Class Server, third version

5. Secrecy-Type System

Prevention of information flow from one information holder to another one with a lower level have been considered in the literature. In our setting of active objects, characteristics such as information hiding and encapsulation imply that there is no external access to class fields [9]. Instead, we need to prevent information flow from one object to another, which we specify by means of static rules for acceptable information flow. We enforce this policy in our secrecy-type system for any well-typed program. Hence, our static analysis does not need to imply any restrictions inside a class such as limitations on information flows from high-level variables to low-level ones (e.g. $v_{High} := v_{Low}$), unless high information can be revealed in communication among objects or during suspension. Therefore, despite the fact that static analysis usually appears as a rather pessimistic and restrictive technique implying a high rate of false positives [17], we are able to be less restrictive. In order to make our static analysis as precise as possible, we allow the secrecy levels of program variables differ with the different program points. This makes our analysis flexible. However, we rely on level information about fields formulated in a way similar to a class invariant, to be respected upon leaving a method invocation.

Our analysis is done class-wise. This is possible in our setting since remote access to fields is forbidden and since all object interaction is done by methods declared in an interface. The secrecy analysis of a class only depends on that class declaration, related interfaces, and the class parameter declaration of instantiated classes (through the **new** construct). We assume a well-typed program and assume each method call $e.m(\overline{e})$ is augmented by annotating the method name m by the interface of the callee e (as in $e.m_I(\overline{e})$), or the enclosing class when e is this. The secrecy-type system for classes and methods are shown in Fig. 8. The confidentiality of a class definition Cl is formalized by judgments of the form

$$\vdash\ Cl\ \ \mathbf{ok}$$

expressing that the class definition obeys the confidentiality rules. The confidentiality of a method definition M is formalized by judgments of the form

$$C \vdash M\ \mathbf{ok}$$

where C is the enclosing class. The confidentiality of a *statement* s is formulated by considering judgments of the form

$$C \vdash [\Gamma, pc]\ s\ [\Gamma', pc']$$

where C is the enclosing class, Γ is a mapping binding variable names to confidentiality levels for a given program point, and pc is the confidentiality level of the current program point. As Γ and pc depends on the program point, we let the "pre-binding" $[\Gamma, pc]$ denote the bindings in the pre-state of s and the "post-binding" $[\Gamma', pc']$ those in the post-state of s. Finally, the confidentiality of expressions and right-hand-sides rhs, given in Fig. 9, are formulated by judgments of the form

$$C \vdash [\Gamma, pc]\ rhs :: l$$

where l is the resulting confidentiality level of rhs. Post-bindings are not needed here as our expressions and righ-hand-sides are side-effect free.

For simplicity, we let the mapping Γ_C (corresponding to table look-up) represent the *declared* secrecy levels of fields and class parameters for a class C as given in the class definition. If the secrecy level of a field w is declared as l, the binding $w \mapsto l$ is included in Γ_C. In contrast, Γ expresses confidentiality information depending on a particular program point. Since Γ-levels of class fields can increase and decrease, the type rules insist that at the end of each method (and at each ion point) their resulting levels should not exceed the declared secrecy level (or equal). This allows us to assume the declared level at each method start and after suspension. Furthermore, the notation $\Lambda[I, m, i]$ denotes the level of the ith parameter of the method as *declared* in interface I. And similarly for classes. For a class C, we let C also be the name of the class constructor (initialization code).

According to Rule S-CLASS in Fig. 8, confidentiality of each class is satisfied, or simply is *ok*, if the confidentiality of each method is satisfied. The confidentiality of a method (see Rule S-METHOD) is

$$\text{(S-CLASS)}\quad \frac{C \vdash\ M_i\ \mathtt{ok},\quad \text{for each } M_i \in \overline{M}}{\vdash \mathtt{class}\ \ C(\overline{cp} : \overline{U})\{\overline{w} : \overline{U}';\ \ \overline{M}\}\ \mathtt{ok}}$$

$$\text{(S-METHOD)}\quad \frac{\begin{array}{c} C \vdash [\Gamma_C[\overline{y} \mapsto \mathcal{L}(\overline{U}), \overline{x} \mapsto \overline{\mathsf{Low}}], \mathsf{Low}]\ \ s\ \ [\Gamma, pc] \\ C \vdash [\Gamma, pc]\ \ e :: l' \\ l' \sqsubseteq l \\ \Gamma[\overline{w}] \sqsubseteq \Gamma_C[\overline{w}] \end{array}}{C \vdash T : l\ \ m(\overline{y} : \overline{U})\{\mathtt{var}\ \ \overline{x} : \overline{T}; s; \mathtt{return}\ \ e\}\ \mathtt{ok}}$$

Figure 8: SeCreol confidentiality type system for classes and methods where Γ_C denotes the declared secrecy levels for class parameters and fields, in class C and Γ expresses confidentiality information depending on a particular program point.

Map notation A mapping M is given by a set of bindings $z_i \mapsto value_i$ for a finite set of disjoint identifiers z_i, the *domain*. The empty map is denoted $\emptyset$. Map look-up is written $M[z]$ where z is an identifier. A map update, written $M[z \mapsto d]$, is the map M updated by binding z to d, regardless of any previous bindings of z. Similarly $M[S]$ denotes M updated with a set S of (disjoint) bindings.

satisfied if its body satisfies confidentiality starting with the declared level bindings (for fields and class parameters, method parameters, and local variables) and with Low as the starting pc level, and resulting in some binding $[\Gamma, pc]$ such that Γ respects the declared field and class parameter bindings levels (i.e., $\Gamma[z] \sqsubseteq \Gamma_C[z]$ for each field/class-parameter z) and such that the returned value respects the declared output level of the method. As stated before, we check $\Gamma[z] \sqsubseteq \Gamma_C[z]$ because the secrecy level of program variables is allowed to be changed in different program points. This is not unlike previous approaches such as [10], except that we make no distinction between confidential and non-confidential variables as long as they do not affect the communication behavior. However, a complication for object oriented programs is that the order of method calls (and suspended processes) is not statically given, therefore the declared level of fields (and class parameters) must be respected at each method return and suspension point. This allows us to assume the declared level at method start and after suspension. This implies that the level of a field may temporarily be higher than the declared level. As we will explain with more details in secrecy-type system, it also implies that a simple fix-point calculation is required to be used when dealing with while-loops.

The SeCreol secrecy-type system for expressions and statements are shown in Fig. 9 and Fig. 10, respectively. These figures present a collection of typing rules describing which secrecy type is assigned to each occurrence of an expression and program variable. In general, based on these rules, the level of an occurrence of an expression is determined using Γ and pc. The rules check that each occurrence of an actual parameter (or return value) respects the declared level of the corresponding formal parameter (or method return level), and that fields and class parameters respect the corresponding declared levels at suspension points and at method returns. In our formalization this is checked by premises in the rules; thus if these premises cannot be derived, the program will not be accepted as a program satisfying the secrecy rules. Note that each statement may adjust Γ, but only **if** and **while** statements may affect pc. Thus the level of variables and pc may differ at different program points, which for example means a call that is acceptable at one program point, might be unacceptable at another point. Furthermore, the rules ensure that parameters of calls made in a branch with a high condition will be high and may therefore not leak information.

Rule S-EXP states that the confidentiality of an expression e is achieved by $\Gamma[e] \sqcup pc$. We include pc since it represents the context level of the current program branch. Thus a low level expression occurring in a program branch with level pc, gets pc as its level, since it may reveal context information. We define $\Gamma[e]$ as follows: For a constant c (including null, this, *void*, and caller) $\Gamma[c]$ is Low (i.e., $\perp$), $\Gamma[e \sqsubseteq e']$ is High (i.e., $\top$), and for other kinds of expressions (including function applications) $\Gamma[e]$ is defined as $\sqcup_{v \in e} \Gamma[v]$, where v ranges over the variables textually occurring in e, and $\Gamma[v]$ is its level recorded in Γ. For simplicity, we here ignore so-called sanitizer functions, which are special functions resulting in a lower level than some of its inputs. Moreover, Object identities are not confidential, thus object variables are typically declared with

$$\text{(S-EXP)}\quad C \vdash [\Gamma, pc]\ e :: \Gamma[e] \sqcup pc \qquad\qquad \text{(S-NEW)}\quad \frac{C \vdash [\Gamma, pc]\ e_i :: l_i \qquad l_i \sqsubseteq \Gamma_{C'}[cp_i]}{C \vdash [\Gamma, pc]\ (\textsf{new}\ C'(\overline{e}) : l) :: pc}$$

$$\text{(S-CALL)}\quad \frac{C \vdash [\Gamma, pc]\ e_i :: l_i \quad l_i \sqsubseteq \Lambda[I, m, i]}{C \vdash [\Gamma, pc]\ e.m_I(\overline{e}) :: \Lambda[I, m] \sqcup pc} \qquad\qquad \text{(S-SELFCALL)}\quad \frac{C \vdash [\Gamma, pc]\ e_i :: l_i \quad l_i \sqsubseteq \Lambda[C, m, i]}{C \vdash [\Gamma, pc]\ \textsf{this}.m(\overline{e}) :: \Lambda[C, m] \sqcup pc}$$

Figure 9: SeCreol secrecy-type system for expressions and right-hand-sides.

$$\text{(S-SKIP)}\quad C \vdash [\Gamma, pc]\ skip\ [\Gamma, pc] \qquad\qquad \text{(S-COMPOSITION)}\quad \frac{C \vdash [\Gamma, pc]\ s_1\ [\Gamma_1, pc_1] \qquad C \vdash [\Gamma_1, pc_1]\ s_2\ [\Gamma_2, pc_2]}{C \vdash [\Gamma, pc]\ s_1; s_2\ [\Gamma_2, pc_2]}$$

$$\text{(S-SIMPLE-CALL)}\quad \frac{C \vdash [\Gamma, pc]\ e.m_I(\overline{e}) :: l}{C \vdash [\Gamma, pc]\ \ e!m_I(\overline{e})\ [\Gamma, pc]} \qquad\qquad \text{(S-RHS)}\quad \frac{C \vdash [\Gamma, pc]\ rhs :: l}{C \vdash [\Gamma, pc]\ \ v := rhs\ [\Gamma[v \mapsto l], pc]}$$

$$\text{(S-AWAIT)}\quad \frac{\begin{array}{c} C \vdash [\Gamma, pc]\ e :: l \\ \Gamma[\overline{w}] \sqsubseteq \Gamma_C[\overline{w}] \end{array}}{C \vdash [\Gamma, pc]\ \texttt{await}\ e\ [\Gamma + \Gamma_C, l]} \qquad\qquad \text{(S-AWAIT-CALL)}\quad \frac{\begin{array}{c} C \vdash [\Gamma, pc]\ rhs :: l \\ \Gamma[\overline{w}] \sqsubseteq \Gamma_C[\overline{w}] \end{array}}{C \vdash [\Gamma, pc]\ \texttt{await}\ v := rhs\ [(\Gamma + \Gamma_C)[v \mapsto l], l]}$$

$$\text{(S-IF)}\quad \frac{\begin{array}{c} C \vdash [\Gamma, pc]\ e :: l \\ C \vdash [\Gamma, l]\ s_1\ [\Gamma_1, pc_1] \\ C \vdash [\Gamma, l]\ s_2\ [\Gamma_2, pc_2] \end{array}}{C \vdash [\Gamma, pc]\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ [\Gamma_1 \sqcup \Gamma_2, pc]} \qquad \text{(S-WHILE)}\quad \frac{\begin{array}{c} C \vdash [\Gamma_i, pc_i]\ e :: l_i \\ C \vdash [\Gamma_i, l_i]\ s\ [\Gamma'_i, pc'_i] \\ \Gamma_{i+1} = \Gamma_i \sqcup \Gamma'_i, \quad pc_{i+1} = pc_i \sqcup pc'_i \end{array}}{C \vdash [\Gamma_1, pc_1]\ \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}\ [FIX_i(\Gamma_i), pc_1]} \quad i = 1, 2, \ldots$$

Figure 10: SeCreol secrecy-type system for statements.

a Low level. However, the level of such variables in Γ is affected by the branch level pc as other program variables. Thus the resulting level of object creation is pc as object identities as such are considered Low. For the right-hand-side of a call or new corresponding other rules in Fig. 9, each actual parameter is required to have a level not exceeding the declared level of the corresponding formal parameter. The resulting level of the call right-hand-side is the declared return level of the method, joined with the current context level pc. We observe that

$$C \vdash [\Gamma, pc]\ rhs :: l \ \Rightarrow\ pc \sqsubseteq l$$

which means the *rhs* level is always at least as high as pc. This fact can easily be proved by looking at each case of an expression or right-hand-side *rhs* according to the SeCreol syntax (Fig. 3).

According to the secrecy-type system for statements in Fig. 10, *skip* does not change anything. Similarly, a simple call does not change Γ nor pc, but the actual parameter levels must respect the declared levels of the corresponding formal parameters (as above). For an assignment, object creation statement, or call, $v := rhs$, with level l for rhs, the level of v in Γ is changed to l, which could imply a downgrade or an upgrade (or no change) of level. The pc is not modified since such a statement is considered efficiently terminating without any branching.

For an **await** statement we must ensure that the declared levels of all fields and class parameters are respected, since the suspension may cause other processes to continue, for which we assume these declared levels. As mentioned, the declared level of fields must be respected at the end/suspension of each process. Levels of local variables will remain after an **await** statement since local variables are not affected by other processes. We therefore use map composition $(+)$ in the post-state of an await to overwrite the levels of fields and class parameters by the declared levels (Γ_C). In the case of a call, the effect of the assignment part is added after the map composition since this assignment happens after suspension. A high await condition may cause implicit leakage, since the presence of high information may be leaked through a low output, for instance await <leaving house>; x.report(true) where report takes low input. Therefore the pc level resulting from an await is that of the await condition/right-hand-side (which is at least as high as the former pc level).

With respect to typing of security levels, a blocking call $v := e.m(\overline{e})$ can be seen as the sequence e!m($\overline{e}$); v:= <returned value>, and the non-blocking call **await** $v := e.m(\overline{e})$ can be seen as the sequence e!m($\overline{e}$); await <long enough>; v:= <returned value>. The rules for blocking and non-blocking calls can be derived from this understanding. Note that an await-statement may affect the pc. A high await condition may reveal secret information that may be leaked. An example could be await <leaving home>; athome:= false. where leaving one's home is considered secret. By raising the level after the high await condition, the athome variable becomes high, and leakage through data values is avoided. Other indirect leakage of an await statement is considered in the next section.

As mentioned an **if** statement may cause implicit leakage of high information, i.e., an **if** statement with a high test may reveal secret information through branches with different low level values communicated to other objects. To avoid this, Rule S-IF lifts the pc level of each branch by the level of the test. This will make all expressions occurring in both branches at least as high as the if-test. Thereby this kind of implicit leakage is avoided. (Note that l is at least as high as pc in the rule.) Since the static analysis does not know which branch is taken at runtime, the resulting value of Γ for each variable is calculated as the highest level of each branch. An **if** statement without an else-branch is like an **if** statement with skip in the else-branch.

The treatment of **while** is similar to an **if** statement without an else-branch, except that the static analysis cannot predict how many times the branch is iterated. Each iteration may lift the levels in Γ or pc. However, a loop will have a finite number of program variables and since there is a finite number of levels, there is a minimal fixpoint reachable in a finite number of approximations (typically i equal to one or two). Rule S-WHILE reflects this fixpoint calculation.

The secrecy typing ensures that there is no flow from high values to low values, and that values evaluated in an if-branch with a high test are high (since they may depend on the test), and similarly for values evaluated after an await with a high test or inside a while-loop with a high test. Thus the values of low variables in any program state do not depend on high inputs. Furthermore, this ensures that for any event generated by o, the values of parameters declared as low do not depend on high inputs. A proof of this based on a semantics that includes runtime secrecy levels, is given in [18], which proves the soundness of the secrecy rules presented here. [3]

The subscription example. The subscription example in Fig. 4 has a straight forward secrecy typing. The secrecy analysis of the while loop needs no iteration to reach the fixpoint. The notify call in class newsprovider (as well as that in SUBSCLIENT) has a high actual parameter (n), which is acceptable since the notify method in Client is declared with a high (formal) parameter. However, the if-statement has a high test, and at the network level the pattern of notify calls could cause network leakage, which we consider in the next section. The High annotations on the objects created in the main class concern the runtime level of these objects, and not the variables declared in the main program. The second and third versions of the example have no additional secrecy challenges, the argument empty is low which is always acceptable.

Another example. A (quasi) example is given in Fig. 11 to illustrate possible changes in the levels of fields (xh and xl) and local variables (x). Level changes are written to the right of each line, not repeating unchanged information. The program satisfies the rules for confidentiality, i.e., the program does not leak information in its explicit output and respects field levels at return/await statements. Note that the lowering of xh was needed to make the *check* call allowed, that the higher level of the x was maintained over the await (since

[3] Alternatively, one could add a level to methods, letting this level be used as the starting pc level of the method body, and require that methods called with high pc must be high. In the subscription example, the local variable u would then be high, the notify call would be accepted, and the secrecy analysis of the while loop would need no iteration (as before). However, this would mean that a high method is not observable and therefore do not cause leakage. All methods called by a high method must also be high, which gives some limitations in what is allowed. If we believe that different notify patterns represent an observable leakage, we can consider the generated pattern and check for leakage. This approach is explained below in Sec. 6.

```
interface Passw{
 Int:Low passw(Int:High x)// store password, return a ref number
 Int:High check(Int:Low x)// check validity of password given ref
}

class TEST(Passw o){
 Int:High xh;
 Int:Low xl;

 Int:High test(Int x){  $xh \mapsto High$ . Note: all others are Low
      xl := x;          $xl \mapsto Low$
      x := xh;          $x \mapsto High$
      xh := xl;         $xh \mapsto Low$ . Note: suspension is ok even with x high
      await <low cond.>; $xh \mapsto High, x \mapsto High$ . Note: all others are Low
      xh := o.passw(x); $xh \mapsto Low$ . Note: the call is ok with x high
      x := o.check(xh); $x \mapsto High$ . Note: the call is ok since xh now is Low
      return x          Note: return is ok with x High, since $xh \sqsubseteq High \wedge xl \sqsubseteq Low$.
}}
```

Figure 11: An example showing level changes in fields and local variables (indicated to the right in each line).

x is local), that the higher level of x was acceptable in the *passw* call, and that the high level of the local variable x is allowed at the return point (after which x is de-allocated). If the await condition had been high, pc would be raised to high after the await, and the call to check would not be secrecy-type correct since xh would then be high.

6. Network Level Leakage

We here consider enforcement of network-level non-interference (INI_o) for SeCreol programs by means of *static trace analysis.* We assume a given program that is secrecy-type correct, i.e., has passed the secrecy-type analysis of Section 5. Moreover, we assume that await-, if-, and while-tests are decorated with the levels resulting from the secrecy-type analysis, using the notation e_l, and we assume the interface of a callee is known from the type analysis.

The static analysis is class-wise and we check that a class is not leaking network information, according to INI_{this}. The analysis is based on *trace expressions*, Δ, detected by means of static analysis applied to *method bodies.* The trace expression of a method reflects the possible traces of an execution of that method including calls generated and consumed by the method. Each trace expression may contain "high" subtraces, caused by high if- and while-conditions. Therefore we check whether any high (sub)trace can be reduced to a low (sub)trace (given the context of the class), as formalized below by *INIcheck.* In the CLASS rule of Fig. 12, the premise $INIcheck(\Delta_{m_i})$ checks the INI property for the trace expression of each method m_i of the class. It must be checked that each trace expression reveals no high information, as detailed further below. We let **isOK** denote that a class declaration satisfies interaction non-interference, and we let M **reveals** Δ denote that a method declaration M reveals the trace set Δ. In the METHOD rule, $default_T$ denotes the default initial value for variables of type T. The substitution of default values for the local variables makes the initial values explicit. In this analysis, we ignore the initial reaction event since it is implicit for the given method.

For statements s we consider judgments of the form $\vdash [\Delta'] \; s \; [\Delta]$. Due to non-determinism caused by suspension and process queues, we cannot estimate the exact history of a method body as a single trace, but may estimate it as an (extended) regular expression, using $(\ldots|\ldots)$ for choice, semicolon for sequential composition, and superscript $*i$ for repetition, and in addition $\bullet$ for any (finite) sequence. The latter

$$\frac{\begin{array}{c}\text{(CLASS)}\\ \vdash m_i(y)\{\texttt{var}\ x; s; \texttt{return}\ e\}\ \texttt{reveals}\ \Delta_{m_i}\ ,\quad \text{for each } m_i \in \overline{M}\\ INIcheck(\Delta_{m_i})\ ,\quad \text{for each } i\end{array}}{\vdash \texttt{class}\ C(cp)\{w; \overline{M}\}\ \texttt{isOK}}$$

$$\frac{\begin{array}{c}\text{(METHOD)}\\ \vdash [\Delta]\ s\ [\mathsf{caller} \leftarrow \mathsf{this}.m]\end{array}}{\vdash m(y)\{\texttt{var}\ T\ x; s; \texttt{return}\ e\}\ \texttt{reveals}\ \Delta[default_T/x]}$$

Figure 12: Network level rules for classes and methods.

represents unknown activities of the object during suspension. Thus a trace expression defines a set of possible traces. Input events will not occur in the trace expressions, since they cannot be detected from the code. But we include reaction events of the form $\mathsf{this} \twoheadleftarrow o.m$ because they can be detected from the code and give implicit information about the corresponding input events ($\mathsf{this} \leftarrow o.m$). So even though reaction events are not directly observable in the class code; they implicitly restrict the time where the corresponding observable input event $\mathsf{this} \leftarrow o.m$ may occur. For instance, a blocking call to m on an external object o gives the trace $\mathsf{this} \rightarrow o.m; \mathsf{this} \twoheadleftarrow o.m$ while a simple call to m gives $\mathsf{this} \rightarrow o.m$. In the first case there will not be any output from the object between the observed events $\mathsf{this} \rightarrow o.m$ and $\mathsf{this} \leftarrow o.m$, as opposed to the second case which gives no restriction for how late the completion event may appear. So an observer may distinguish a difference. Without the reaction events in the traces, the two cases will have the same trace expressions and our system would be unsound since observable differences are not captured. Simple, blocking, and suspending calls are observably different and are represented differently in the traces.

Furthermore, self-calls pose some non-trivial challenges. For instance, consider the code

if $\mathsf{e}_{\mathsf{High}}$ then v:=this.m1() else v:=this.m2() fi

If $m1$ makes the call o!n(true) and $m2$ makes the call o!n(false), where the parameter is Low, the outcome of the high if-test is leaked to object o. In order to handle such cases we include self-calls in the traces even though they are not observable. The example will then not pass the INIcheck test. And in the example if $\mathsf{e}_{\mathsf{High}}$ then this.m(true) else this.m(false) fi , the outcome of the if-test may seem to be leaked if $m(x)$ makes the call $o.n(x)$ where o is an external object. However, in this example the argument to m must be high in order to pass the secrecy-typing requirements, which means that there is no leakage.

Another challenge related to self-calls is that a call $o.m(\overline{e})$ may be a self-call if o equals this, unless the interface of o is not supported by the enclosing class (for instance when m is not implemented in the class). This means that calls can be categorized as self-calls (calls to this), external calls (calls through interfaces not supported by the enclosing class), and calls for which we do not know at static time if they are self-calls or not. The analysis must deal with all these categories. The call events of external calls are visible to an observer, but not for self-calls. However, for a self-call the activity caused by called method might be important, but not for external calls since the output of such an invocation is not an output of this object. Thus the static treatment is non-trivial.

In the analysis of statements, we employ backward trace analysis for detection of generated observable events, by triples $[\Delta']\ s\ [\Delta]$ similar to Hoare triples, where Δ denotes a trace expression. Intuitively, it means that the statement s generates the trace Δ' (the *pre-trace*) when Δ is the trace generated after s has terminated (the *post-trace*). The notation $\Delta[e/x]$ denotes the trace set expression Δ with all occurrences of the variable x replaced by the expression e. The example if high then o:=o'; o!m(...) else o'!m(...) fi, motivates that the effects of assignments should be considered in the analysis (in order to detect non-leakage in this case). As we will explain later in this section, the above code satisfies the policy. Therefore, the reason for doing a backward analysis is that the generated trace expressions contain program variables, and therefore the effect of assignments must be considered. The handling of assignments can then be done by backward substitution as in Hoare logic. Thus an assignment $x := e$ causes the replacement of e for x in

$\vdash [\Delta]\ skip\ [\Delta]$

$\vdash [\Delta[e/x]]\ x := e\ [\Delta]$

$\vdash [(\mathsf{this} \to o.m); \Delta]\ o!m(\overline{e})\ [\Delta]$

$\vdash [(\mathsf{this} \to o.m); (\mathsf{this} \twoheadleftarrow o.m); \Delta[\mathit{fresh}/x]]\ x := o.m(\overline{e})\ [\Delta]$

$\vdash [((\mathsf{this} \to x.C); (\mathsf{this} \twoheadleftarrow x.C); \Delta)[\mathit{fresh}/x]]\ x := \mathbf{new}\, C(\overline{e})\ [\Delta]$

$\vdash [(\bullet)_l; \Delta]\ \mathtt{await}\ e_l\ [\Delta]$

$\vdash [(\mathsf{this} \to o.m); \bullet; (\mathsf{this} \twoheadleftarrow o.m); \Delta[\mathit{fresh}/x]]\ \mathtt{await}\, x := o.m(\overline{e})\ [\Delta]$

Figure 13: Trace axioms for basic statements. Here *fresh* denotes a fresh symbol.

$$\text{(SEQ-COMP)}\quad \frac{\vdash [\Delta'']\ s\ [\Delta'] \qquad \vdash [\Delta']\ s'\ [\Delta]}{\vdash [\Delta'']\ s; s'\ [\Delta]}$$

$$\text{(IF-ELSE)}\quad \frac{\vdash [\Delta_1]\ s_1\ [\varepsilon] \qquad \vdash [\Delta_2]\ s_2\ [\varepsilon]}{\vdash [(\Delta_1 | \Delta_2)_l; \Delta]\ \mathtt{if}\, e_l\, \mathtt{then}\, s_1\, \mathtt{else}\, s_2\, \mathtt{fi}\ [\Delta]}$$

$$\text{(WHILE)}\quad \frac{\vdash [\Delta]\ s\ [\varepsilon]}{\vdash [((\Delta[\overline{\mathit{fresh}/w}])_l)^{*i}; \Delta']\ \mathtt{while}\, e_l\, \mathtt{do}\, s\, \mathtt{od}\ [\Delta']}$$

Figure 14: Rules for trace analysis. In Δ^{*i} each fresh constant c is replaced by c^i, making freshness depend on the iteration, and $\overline{w}$ is the list of program variables used in a right hand side inside an iteration.

the pre-trace, letting $[e/x]$ denote the substitution. Similar substitutions are caused by object creation, blocking, and non-blocking calls with assignment part $x := rhs$, except that here the value assigned to x is not statically given, and is reflected by a fresh value.

In the axioms of Fig. 13 for basic statements, the pre-trace Δ' is expressed by means of the post-trace, given by a symbol Δ, consistent with left-constructive analysis. Based on these rules, *skip* does not have any effect on a trace while in case of an assignment, the pre-trace is determined by replacement of x with e in the post-trace. Moreover, in case of a simple call, a call event is added to the pre-trace (even if it is a self-call). In the rules for **await**, the symbol $\bullet$ represents arbitrary traces caused by suspension. Since a high test in a conditional await-statement may depend on high variables, its enabledness may reveal secret information. For instance, an await statement with a condition testing *raised privileges* gives a high (sub)trace. Therefore the $\bullet$ is considered high in this case, which affects the INIcheck. For instance, a program path going through a conditional await testing *raised privileges* gives a high trace expression, which cannot be used to match any low trace. The notation of $\Delta[\mathit{fresh}/x]$ in these rules denotes that all occurrences of the variable x is replaced by a fresh constant.

Rules for trace analysis are shown in Fig. 14. The rule for sequential composition resembles that of Hoare Logic. We may for instance derive $\vdash [\Delta']\ skip; s\ [\Delta]$ from $\vdash [\Delta']\ s\ [\Delta]$. In Rule IF-ELSE we encode the traces of the branches, Δ_1 and Δ_2, into a branching expression, $(\Delta_1 \mid \Delta_2)_l$ where l is the level of the if-expression (as obtained by the secrecy-typing). The rule for while is similar, using superscript $*i$ for repetition where the *iteration index* i allows us to refer to each iteration. In particular, this allows freshness to be dependent on an iteration i, as captured by Δ^{*i}. Remark that the nesting of if- and while-statements determine the inner secrecy labels in a regular expression. For a loop with a counter variable j starting on 1 and such that $j := j + 1$ occurs in the loop body as the only update of j, we simply use j as the iteration index, ignoring the statement $j := j + 1$ in the further analysis, and avoiding replacing j with a fresh constant, and thereby

$(\Delta \mid \Delta)_l$	$\longrightarrow$	Δ	
$\bullet;\bullet$	$\longrightarrow$	$\bullet$	
$(\varepsilon)_l$	$\longrightarrow$	ε	
$\mathsf{this} \rightarrow \mathsf{this}.m;\Delta$	$\longrightarrow$	Δ	for a simple, recursive call to m outside a branch, if Δ does not start with $[\bullet;]\mathsf{this} \twoheadleftarrow \mathsf{this}.m$

Figure 15: Simplification rules for the trace set of a method m. A self-call $\mathsf{this} \rightarrow \mathsf{this}.m$ is detected as simple if it is not followed by $\mathsf{this} \twoheadleftarrow \mathsf{this}.m$ nor $\bullet;\mathsf{this} \twoheadleftarrow \mathsf{this}.m$. And a self-call is recursive if it is to the enclosing method m (as in method *mask*).

allowing more simplifications.

Subscription example. Consider the original notify method defined in Fig. 4. We need to find Δ' such that $[\Delta']$ *body* $[\mathsf{caller} \leftarrow \mathsf{this}.\mathit{notify}]$ for the *body*. Using the rules (in a left to right manner) we determine Δ' as

$$((\mathsf{this} \rightarrow \mathit{users}[i].\mathit{notify} \mid \varepsilon)_{\mathsf{High}})^{*i};\mathsf{caller} \leftarrow \mathsf{this}.\mathit{notify}$$

which contains high subtraces. As explained below, it will not satisfy the INIcheck.

For the second version of notify, we get the trace expression

$$((\mathsf{this} \rightarrow \mathit{users}[i].\mathit{notify} \mid \mathsf{this} \rightarrow \mathit{users}[i].\mathit{notify})_{\mathsf{High}})^{*i};\mathsf{caller} \leftarrow \mathsf{this}.\mathit{notify}$$

As explained below it simplifies to $(\mathsf{this} \rightarrow \mathit{users}[i].\mathit{notify})^{*i};\mathsf{caller} \leftarrow \mathsf{this}.\mathit{notify}$, which does not contain high subtraces, and satisfies the INI property.

For method make_subscr, we need to solve $[\Delta']$ *body* $[\mathsf{caller} \leftarrow \mathsf{this}.\mathit{make\_subscr}]$ for the method *body*. We determine Δ' as

$$\mathsf{this} \rightarrow s.\mathit{subscr};\bullet;\mathsf{this} \twoheadleftarrow s.\mathit{subscr};\mathsf{caller} \leftarrow \mathsf{this}.\mathit{make\_subscr}$$

which has no high subtrace, and is therefore not causing network leakage.

INIcheck. The $INIcheck$ test of a class is done by checking $INIcheck(\Delta_m)$ for each method m of the class (including the constructor) where Δ_m is the trace expression generated for the body of m. The test is passed if Δ_m can be reduced to a trace expression without high subtraces, using the simplification rules defined in Fig. 15. If the simplified Δ_m has high subtraces, we flatten Δ_m to a set of trace expressions t_i, by flatting the branches, where each t_i is defined as high if it goes through a path of Δ_m with a high branch (or subtrace), and otherwise low. For example, a trace such as $\Delta_1;(\Delta|\Delta')_l;\Delta_2$ is flattened to $t_1 = \Delta_1;\Delta_l;\Delta_2$ and $t_2 = \Delta_1;\Delta'_l;\Delta_2$. For each flattened high trace t_i we must then check if it can be recreated from the set of flattened low traces of the same method, considering also possible other activities during suspension. This check is denoted

$$t_i \;\mathtt{\textbf{matches}}\; S$$

where S is the union of the set of *low traces of the same method* and the set of *low traces of any background self activity* without the final non-observable completion events. The background self activity is captured by the set of flattened traces of recursive methods, with a simple or non-blocking recursive self-call, and where the method is called by a simple self-call from the class constructor (directly or indirectly). The rules for detecting recreation is defined in Fig. 16. Thus a high trace t_i passes the check if t_i **`matches`** S following from the rules, where as mentioned, S is the union of all low traces of the same method and low traces representing self behavior. The final non-observable completion event is omitted in a trace representing self behavior since this event is non-observable. Clearly, in order to match the final completion event of t_i one must involve a low trace of the same method, while self behavior may appear at suspension points.

The simplification rules in Fig. 15 are used to reduce an INIcheck. The rules are confluent and terminating. The first rule says that a branch expression with two identical branches can be simplified, removing

ε	**matches**	S		
t	**matches**	S	**if**	$t \in S$
$t; t'$	**matches**	S	**if**	t **matches** $S \wedge t'$ **matches** $S \cup \mathit{sscalls}(t)$
$t; t''$	**matches**	S	**if**	$t; \bullet; t'$ **matches** $S \wedge t''$ **matches** $S \cup \{t'\} \cup \mathit{sscalls}(t)$
$\bullet; t$	**matches**	S	**if**	t **matches** S
$t; \bullet$	**matches**	S	**if**	t **matches** S
$t; t''$	**matches**	S	**if**	$t; (t')^{*i}; t''$ **matches** S
$t; t'; t''$	**matches**	S	**if**	$t; (t')^{*i}; t''$ **matches** S
$(t; t')^{*i}$	**matches**	S	**if**	$(t; \bullet; t')^{*i}$ **matches** S

Figure 16: Rules to determine if a (high) trace matches a set of low traces, while adding any new suspended processes to S. t is a branch-free trace expression, and S a set of such expressions. Here $\mathit{sscalls}(t)$ denotes the set of low trace expressions, without the final non-observable completion event, for each method with a simple self-call in t. (Exemplified in Fig. 17).

the level of the branch expression. The second rule says that suspending twice in a row is equivalent to suspending once (since any number of enabled suspended processes may be taken during each suspension). The last rule says that a simple recursive call (after the last suspension point of a trace) can be ignored since the event is non-observable, the recursive invocation will be done later during suspension, and since an irreducible trace must be revealed in a single invocation. Thus a leakage in a recursive method can be found in the body without the recursive call. A call is detected as a self-call if it has the form $\mathsf{this} \rightarrow \mathsf{this}.m$ and as recursive if m is to the enclosing method. An event $\mathsf{this} \rightarrow \mathsf{this}.m$ in a flattened trace is detected as a simple self-call if it is not followed by $\mathsf{this} \twoheadleftarrow \mathsf{this}.m$ nor $\bullet; \mathsf{this} \twoheadleftarrow \mathsf{this}.m$. In the rules we let $\mathit{sscalls}(t)$ denote the low traces of the methods called by simple self-calls in trace t, omitting the final completion event ($\mathsf{this} \leftarrow \mathsf{this}$), which is non-observable. In order to limit the amount of false positives, one may add further rules, such as replacing $(\Delta|\Delta')_l$ by $(\Delta'|\Delta)_l$ according to some ordering over trace expressions. And one may add that an initial $\bullet$ in a branch can be removed, if the branch expression is preceded by a $\bullet$, and similarly for a final $\bullet$ in a branch. However, a more detailed study is beyond the scope of this paper.

The rules in Fig. 16 incorporate suspended behavior S of an object, by starting with a (low and flattened) trace t in S and either stopping at a suspension point ($\bullet$), adding the remaining part of t to S, and continuing with another trace in S. We may add a suspension point at the front or at the end of trace in S, since each such trace is starting from and ending in a suspension (explaining rule 5 and 6 of Fig. 16). And an iteration may be included or skipped, and a bullet in an iteration may be ignored. Thus S is in general infinite; however, in the context of checking whether a given flattened trace expression t_i is in S, we may expand S while matching t_i from left to right. This can be done in finite time letting each application of a rule match a lager part of t_i.

The formalization of matching depends on the *duration of network observations*. If we assume the observations are made over a short term, it is plausible that any method of an interface of the class has been called before the observations starts, and its low traces should therefore be included in S. If the observation is long term which we take as the default, this assumption is not appropriate, and the low traces of the active self behavior may safely be included in S. Thus we consider here the worst case in this respect, but the same approach can be used for the case of observations with short duration.

Examples. Fig. 17 explains the application of rules in INIcheck with some synthetic examples to cover different possibilities. In the first example, the if statement has the trace $(\mathit{this} \rightarrow o'.m\,; \mathit{this} \twoheadleftarrow o'.m \mid \mathit{this} \rightarrow o'.m; \mathit{this} \twoheadleftarrow o'.m)_{\mathsf{High}}$ simplified to $\mathit{this} \rightarrow o'.m; \mathit{this} \twoheadleftarrow o'.m$ using the simplification rules, which has no high subtrace, and therefore there is no network leakage. $\mathit{Exp}2$ will not pass the INIcheck because $m1$ is textually different from $m2$. Remark that if $o = \mathsf{this}$ and if $m2$ and $m1$ make the same calls, we have a false positive. In the third example, there is no high subtrace, and therefore no leakage. The examples $\mathit{Exp}4$ to $\mathit{Exp}7$ do not satisfy the INIcheck because they have a pair of high traces (and here no background self activity set S is given). In $\mathit{Exp}7$ the two self-calls may indirectly cause an observable difference, if m has observable output, since the call in the then-branch makes this output happen before the current method

$$\begin{array}{c}
(\textsc{Exp}1)\\
\mathtt{if}\ e\ \mathtt{then}\ o := o';\ v := o.m\\
\mathtt{else}\ v := o'.m\ \mathtt{fi}\\ \hline
\checkmark : \text{Trace: } (\mathit{this} \to o'.m\ ; \mathit{this} \twoheadleftarrow o'.m\\
\mid \mathit{this} \to o'.m; \mathit{this} \twoheadleftarrow o'.m)_{\mathsf{High}}\\
\text{Simplified to } \mathit{this} \to o'.m; \mathit{this} \twoheadleftarrow o'.m\\
\text{No high subtrace.}
\end{array}$$

$$\begin{array}{c}
(\textsc{Exp}2)\\
\mathtt{if}\ e\ \mathtt{then}\ v := o.m_1\\
\mathtt{else}\ v := o.m_2\ \mathtt{fi}\\ \hline
\times : \text{Trace: } (\overbrace{\mathit{this} \to o.m_1; \mathit{this} \twoheadleftarrow o.m_1}^{t_1}\\
\mid \overbrace{\mathit{this} \to o.m_2; \mathit{this} \twoheadleftarrow o.m_2}^{t_2})_{\mathsf{High}}\\
\text{Both } t_1, t_2 \text{ are high.}
\end{array}$$

$$\begin{array}{c}
(\textsc{Exp}3)\\
v := o.m_1;\\
\mathtt{await}\ e_{\mathsf{Low}};\\
v := o.m_2\\ \hline
\checkmark : \mathit{Trace} : \mathit{this} \to o.m_1; \mathit{this} \twoheadleftarrow o.m_1;\\
\bullet; \mathit{this} \to o.m_2; \mathit{this} \twoheadleftarrow o.m_2\\
\text{No high subtrace.}
\end{array}$$

$$\begin{array}{c}
(\textsc{Exp}4)\\
\mathtt{if}\ e\ \mathtt{then}\ v := o.m\\
\mathtt{else\ await}\ v := o.m\ \mathtt{fi}\\ \hline
\times : \mathit{Trace} : (\overbrace{\mathit{this} \to o.m; \mathit{this} \twoheadleftarrow o.m}^{t_1}\\
\mid \overbrace{\mathit{this} \to o.m; \bullet; \mathit{this} \twoheadleftarrow o.m}^{t_2})_{\mathsf{High}}\\
\text{No simplification, and } t_1, t_2 \text{ high.}
\end{array}$$

$$\begin{array}{c}
(\textsc{Exp}5)\\
\mathtt{if}\ e\ \mathtt{then}\ v := o.m\\
\mathtt{else}\ o!m\ \mathtt{fi}\\ \hline
\times : \mathit{Trace} : (\overbrace{\mathit{this} \to o.m; \mathit{this} \twoheadleftarrow o.m}^{t_1}\\
\mid \overbrace{\mathit{this} \to o.m}^{t_2})_{\mathsf{High}},\ t_1, t_2 \text{ both high.}
\end{array}$$

$$\begin{array}{c}
(\textsc{Exp}6)\\
\mathtt{if}\ e\ \ \mathtt{then\ await}\ v := o.m\\
\mathtt{else}\ o!m\ \mathtt{fi}\\ \hline
\times : \text{Trace: } (\mathit{this} \to o.m; \bullet; \mathit{this} \twoheadleftarrow o.m\\
\mid \mathit{this} \to o.m)_{\mathsf{High}}
\end{array}$$

$$\begin{array}{c}
(\textsc{Exp}7)\\
\mathtt{if}\ e\ \ \mathtt{then\ await}\ v := \mathit{this}.m\\
\mathtt{else}\ \mathit{this}!m\ \mathtt{fi}\\ \hline
\times : \mathit{Trace} : (\mathit{this} \to \mathit{this}.m; \bullet; \mathit{this} \twoheadleftarrow \mathit{this}.m\\
\mid \mathit{this} \to \mathit{this}.m)_{\mathsf{High}}
\end{array}$$

$$\begin{array}{c}
(\textsc{Exp}8)\\
\mathit{mtd}_1()\{\\
\mathtt{if}\ (e)_{Low}\ \mathtt{then}\ v := o.m_1\ \mathtt{fi};\\
\mathtt{if}\ (e')_{Low}\ \mathtt{then}\ v := o.m_2\ \mathtt{fi};\\
\mathtt{if}\ (e)_{High}\ \mathtt{then}\ v := o.m_1\\
\mathtt{else}\ v := o.m_2\ \mathtt{fi}\}\\ \hline
\times : \text{Traces similar to } EXP2.
\end{array}$$

$$\begin{array}{c}
(\textsc{Exp}9)\\
\mathit{constructor}()\{\mathit{this}!n()\}\\
n()\{v := o.m1; \mathit{this}!n\}\\
\mathit{mtd}_1()\{\mathtt{if}\ e\ \mathtt{then}\ v := o.m_1\ \mathtt{else}\ \mathit{skip}\ \mathtt{fi}\}\\ \hline
\checkmark : \text{Trace of } \mathit{mtd}_1 : (\mathit{this} \to o.m_1; \mathit{this} \twoheadleftarrow o.m_1 \mid \varepsilon)_{\mathsf{High}}; \mathit{caller} \leftarrow \mathit{this}.\mathit{mtd}_1\\
\text{Flattened} : t_1 = (\mathit{this} \to o.m_1; \mathit{this} \twoheadleftarrow o.m_1; \mathit{caller} \leftarrow \mathit{this}.\mathit{mtd}_1)_{\mathsf{High}} \text{ and } t_2 = \mathit{caller} \leftarrow \mathit{this}.\mathit{mtd}_1\\
\text{Simplified trace of method } n :\ \ t_n = \mathit{this} \to o.m_1; \mathit{this} \twoheadleftarrow o.m_1\\
\text{Here } t_1 = t_n; t_2 \text{ and } t_2 \text{ is low. So } t_1\ \mathtt{matches}\ S \text{ since both } t_1, t_n \text{ are in } S.
\end{array}$$

$$\begin{array}{c}
(\textsc{Exp}10)\\
n()\{v := o.m1\}\\
\mathit{mtd}_1()\{\mathit{this}!n(); \mathtt{await}\ e_{\mathsf{Low}}; \mathtt{if}\ e\ \mathtt{then}\ v := o.m_1\ \mathtt{else}\ \mathit{skip}\ \mathtt{fi}\}\\ \hline
\checkmark : \text{Trace of } \mathit{mtd}_1 : (\mathit{this} \to \mathit{this}.n; \bullet; (\mathit{this} \to o.m_1; \mathit{this} \twoheadleftarrow o.m_1 \mid \varepsilon)_{\mathsf{High}}; \mathit{caller} \leftarrow \mathit{this}.\mathit{mtd}_1)\\
\text{Flattened} : t_1 = \mathit{this} \to \mathit{this}.n; \bullet; (\mathit{this} \to o.m_1; \mathit{this} \twoheadleftarrow o.m_1)_{\mathsf{High}}; \mathit{caller} \leftarrow \mathit{this}.\mathit{mtd}_1\\
\text{and } t_2 = \mathit{this} \to \mathit{this}.n; \bullet; \mathit{caller} \leftarrow \mathit{this}.\mathit{mtd}_1\\
\text{Trace of } n :\ \ t_n = \mathit{this} \to o.m_1; \mathit{this} \twoheadleftarrow o.m_1; \mathit{caller} \leftarrow \mathsf{this}.n\\
\text{Thus } \mathit{sscalls}(\mathit{this} \to \mathit{this}.n) = t'_n = \mathit{this} \to o.m_1; \mathit{this} \twoheadleftarrow o.m_1\\
\text{Then } \mathit{this} \to \mathit{this}.n; t'_n; \mathit{caller} \leftarrow \mathit{this}.\mathit{mtd}_1\ \mathtt{matches}\ S \text{ since } t_2 \in S \text{ and } t'_n \text{ is in the extended } S.
\end{array}$$

Figure 17: Examples of network level rule applications for INIcheck. Here e is high and ✓ indicates success and × failure.

is finished (with a visible $\leftarrow$ this event) while this need not to be the case for the call in the else-branch. *Exp*8 has a possibility of leakage, which is detected. This happens when more than one call happens at runtime, in which case the second call reveals information about the high test. Otherwise, an attacker cannot distinguish between the high or low if-tests. Moreover, as explained in the details in the figure, *Exp*9 satisfies the INIcheck because the high trace can be matched by low traces, and thus the execution traces are indistinguishable from the attacker's point of view. The last non-trivial example, *Exp*10, also satisfies the INIcheck. It is because the only high flattened subtrace in mtd_1, i.e., t_1 can be also recreated with the combination of the low trace of the same method, i.e., t_2 and the trace of simple self-call to method n in mtd_1, i.e., t_n. In other words, the observer cannot distinguish t_1 from an execution that includes t_2 such that t_n happens at the suspension point. This example shows the application of $sscalls(t)$ as the set of low trace expressions for a method with a simple call in the matching rules in Fig. 16.

The subscription example revisited. The original notify method reveals

$$(\mathsf{this} \to users[i].notify \mid \varepsilon)^{*i}_{\mathsf{High}}; caller \leftarrow \mathsf{this}.notify$$

where i is the iteration index. In order to remove the first high call we need to look at any other background activity in this object. In the first version we do not have any such activity, so based on our simplification rules, the call in the high branch cannot be removed, and not satisfaction on INIcheck is detected. However, for the second version, the redefined notify method reveals the following trace expression

$$(\mathsf{this} \to users[i].notify \mid \mathsf{this} \to users[i].notify)^{*i}_{\mathsf{High}}; caller \leftarrow \mathsf{this}.notify$$

which is simplified to

$$(\mathsf{this} \to users[i].notify)^{*i}; caller \leftarrow \mathsf{this}.notify$$

which means there is no leaking because due to textually equivalence of the two branches the high subscript is removed. Therefore, this method passes the INIcheck test.

For the last version of the subscription example, the redefined version of *notify* reveals the trace expression

$$(\varepsilon \mid (\mathsf{this} \to users[i].notify)^{*i})_{\mathsf{High}}; caller \leftarrow \mathsf{this}.notify$$

Flattening gives the low trace $caller \leftarrow \mathsf{this}.notify$, denoted t_1, and the high trace t_2 given by:

$$((\mathsf{this} \to users[i].notify)^{*i}; caller \leftarrow \mathsf{this}.notify)_{\mathsf{High}}$$

We then need to show that t_2 **matches** $\{t_1\} \cup S$ where S is the low traces of the background self activity. Since t_1 is without suspension, t_2 must end with t_1, and we need to show $(\mathsf{this} \to users[i].notify)^{*i}$ **matches** S. The constructor of class SUBSCLIENT has a simple self-call to *mask* and *mask* has a simple recursive self-call. Thus S contains the trace expression of *mask* (ignoring the final non-observable completion event)

$$(\mathsf{this} \to users[i].notify; \bullet)^{*i}_{\mathsf{Low}}; \mathsf{this} \to \mathsf{this}.mask$$

where the simple recursive self-call can be removed based on the simplification rules. Thus we have

$$(\mathsf{this} \to users[i].notify; \bullet)^{*i} \ \mathbf{matches}\ S$$

Clearly, we also have $(\mathsf{this} \to users[i].notify)^{*i}$ **matches** S by the last rule of Fig. 16. Therefore also this example passes the INIcheck test! In contrast, the original notify method would need a masking method that selects a subset of the users for notification for instance by using non-deterministic choice (if added to the language).

7. Theoretical Results

In this section, we prove that by applying the proposed network trace analysis in Section 6, any possible deviation from the INI policy defined in Section 3 will be detected. The possible execution traces σ for our language are defined by the operational semantics in Appendix A.

For an execution trace σ, the subtrace involving a given object o, denoted σ/o, consists of output events SND_o, input events RCV_o, and internal (input) events RAC_o. The rules for generating trace expressions Δ of an object o talk about the events generated by o, namely SND_o and RAC_o, as well as $\bullet$. Non-observable self calls are included in the traces σ and trace expressions Δ.

Lemma 1 (Traces correspond to the operational semantics)**.** *Consider a trace expression generated by the trace analysis of a given method m. The trace expression will cover all traces possible by an execution of m according to the operational semantics (Appendix A) in the sense that each execution of m gives a trace that is an instantiation of one of the flattened trace expressions, when restricted to events generated by the method execution.*

Proof outline. Consider a given class C and method m with method body s. We show that the trace expression Δ generated for m, based on the trace axioms (TAx) of Fig. 13 and the trace analysis rules (TSt) of Fig. 14, is according to the operational semantics, in the sense that the events generated at runtime by any invocation of m is an instantiation of one of the flattened traces of Δ (instantiating the variables in Δ). To obtain the subtrace generated by an invocation of m we let the events generated by the rules of the operational semantics be tagged by the unique identity of the invocation, given by the value of $\delta[\mathit{callId}]$ where δ is the state of the object in the left-hand-side of the rule. In the trace generated by an execution we may then extract the subtrace with a given *callId* tag, called an invocation trace, which will consist of output and reaction events generated by the object (i.e., SND_o and RAC_o events) with the completion event of the method as the last event.

Consider an arbitrary invocation trace of an arbitrary execution of the given method m. We may then prove that the invocation trace is equal to an instantiation of a flattened trace expression. Each triple $[\Delta]\ s\ [\Delta']$ in the trace analysis can be understood as the Hoare triple $[h \in \{\Delta\}]\ s\ [h \in \{\Delta'\}]$ where h is the local communication history (trace), using for instance a reasoning system similar to [19] (without futures). The soundness of the axioms and rules of Figs. 13 and 14 follows from the soundness of the reasoning system for histories in [19], using this translation to Hoare logic. In particular, the events generated in the pre-traces in the axioms of Fig. 13 correspond exactly to the events generated in the operational semantics, and the substitutions in the assignment-like statements in Fig. 13 correspond to those of Hoare logic. $\square$

Theorem 1 (INI Deviation Detection)**.** *The interaction non-interference policy is satisfied for a set of objects if the corresponding INIcheck is satisfied by the corresponding classes of those objects. For each object o of a class C we have*

$$INIcheck_C \Rightarrow INI_o$$

Proof outline. Let us prove $INIcheck_C \Rightarrow INI_o$ by contradiction, i.e., $INIcheck_C \wedge \neg INI_o$. We assume that there is a class C that based on the rules in Fig. 12 satisfies **`isOK`** and that there is an object o of that class that does not satisfy INI, i.e., $\neg INI_o$. This means that for each method m in that class with s as the method body, Δ is calculated in the form of $[\Delta]\ s\ [\mathit{caller} \leftarrow \mathsf{this}.m]$ based on Fig. 13 and 14. According to Lemma 1, Δ reflects the communication traces obtained from the operational semantics. Due to the satisfaction of INIcheck, we know that each high trace t obtained after simplification and flattening of Δ (using the simplification rules in Fig. 15) can be recreated by the set S of low traces of m and background self activity (i.e., t **`matches`** S), using the rules in Fig. 16.

And, based on $\neg INI$, there are sequences of events σ and σ', and some i, such that:

$$\begin{gathered}(\sigma/o)|i =_L (\sigma'/o)|i \wedge (\sigma/o)[i+1] \in SND_o \wedge \\ \nexists \sigma'' \,.\, (\sigma'/o)|i \leq \sigma'' \wedge (\sigma/o)|i+1 \approx_{net} (\sigma''/o)|i+1\end{gathered}$$

where σ, σ' and σ'' range over possible execution traces. Therefore, based on $\neg INI$ there is not any execution trace σ'' which contradicts that the flattened t is matched by the set S of suspension behaviors. The set of trace expressions representing background self activity provides an underestimation of the process queue of o when an invocation of m is made. Thus there is an execution that continues with an instance of t after $(\sigma'/o)|i$.

It suffices to consider σ and σ' such that the inputs to o from other objects come as late as possible, i.e., a call $o' \rightarrow o$ comes just before $o' \twoheadrightarrow o$ and such that a completion $o \leftarrow o'$ comes just before $o \twoheadleftarrow o'$ (for o' different from o). This can be made possible by considering executions where the objects generating calls or completions to o are slowed down. This has no effect on the behavior of o, and it allows us to derive the missing RCV_o events from a flattened trace expression.

It is also clear that if for each object o_i of class C_i the trace analysis of C_i is OK, and thus INI_{o_i} is satisfied, then for a set of such objects O, INI_O is satisfied as well. Since any receive event for an object must happen after the corresponding send event, we consider the subset of executions where the inner send events match the corresponding receive events. We may assume σ is in this set. Formally, consider any σ, σ' (in this set), and i. We may assume the left side of the implication, i.e. $(\sigma/O)|i =_L (\sigma'/O)|i \wedge (\sigma/O)[i+1] \in SND_O$ and need to prove $\exists\sigma''\,.\,(\sigma'/O)|i \leq (\sigma''/O) \wedge (\sigma/O)|i+1 \approx_{net} (\sigma''/O)|i+1$. We have that $(\sigma/O)[i+1] \in \mathit{SND}_O$ and thus there must be an object o in O such that $(\sigma/o)[i+1] \in \mathit{SND}_o$. This means that we can apply INI_o since σ/o and σ'/o are determined by σ/O and σ'/O and we may choose σ'' such that o is scheduled in the next step (as in σ). $\square$

8. Related Work

As stated in the introduction, programming languages can provide fine-grained control for security issues, and a large amount of work is based on Denning's paper on information flow security [20]. Substantial contributions have been made to prevent disclosure of confidential information based on static, dynamic, or hybrid *information flow control* approaches.

Statically checking information flow to protect confidentiality and integrity is a promising technique as it provides increased precision [20] and low runtime overhead of dynamic security classes [21]. To enforce information flow control policies using static program analysis, program elements are annotated with necessary information. Volpano et al [22] were the first ones to formulate Denning's approach [20] based on program certification, as a type system and proved soundness of a version of non-interference theorem for a core deterministic language. To track information flow in Java, Myer [21] extended the type system of Java and proposed a *decentralized label model*. The extended type system was later implemented in Jif compiler. In another adaptation, type system of functional language OCaml, was extended to Flow Caml [23] by annotating ML types with security labels. Major challenges pertaining to static approaches are that they usually require complicated type annotations and often result in a significant degree of false positives [24]. Although, theorem proving techniques are used in [25, 26] to improve precision of static program analysis.

In contrast, dynamic mechanisms such as [27, 28] are more permissive, imposing high overhead and may require changes to the runtime systems, e.g. special schedulers. Additionally, in a majority of real time systems, security policies vary dynamically [29] and cannot be determined at compile time. In [29] Zheng et al expanded the scope of information flow control by providing mechanism to update label values of program elements during run time. Tse and Zdancewic facilitate dynamic flow control by proving non-interference for a security-typed lambda calculus with runtime principals and enable more expressive security polices [30]. Sabelfeld and Russo [31] compare and contrast static with dynamic program analysis, and deduce (using simple imperative language) that both techniques assure the same level of termination-insensitive non-interference.

Exploration of both static and dynamic approaches are made in [32], and hybrid mechanisms such as [33, 34, 35], are provided to enhance the information flow capability and increasing permissiveness, by realizing static analysis by security type systems and realizing dynamic analysis by monitors. This hybrid approach was also employed in the development of a new system and language, Fabric [36], which is used to build secure distributed information systems. Fusion of static and dynamic mechanisms of analysis for concurrent programs has been proposed by Guernic [37], using an automaton to monitor the information flow for a single execution of a concurrent program. Most of the work in programming language research that provides information flow control is based on the principle of *non-interference*.

M. Miller [38] explores language-based capabilities in the context of the object capability model in his Ph.D. thesis. This model is useful for investigating object communication and computation aspects. However, it focuses on robustness issues rather than security issues. Additionally, Hammer and Snelting [39] propose information flow control based on program dependency graphs, and demonstrates significant reduction in annotation overhead and improved program analysis precision [40].

Our proposed approach falls in the category of static analysis. However, in this approach we have prevented a high false positive rate since, due to hiding and encapsulation in our distributed object-oriented setting, we do not impose unnecessary restrictions on information flow inside objects. In addition to a new security type- and effect-system for the considered language, we propose a new kind of class-wise trace analysis to restrict the flow of control among objects communicating by asynchronous methods, to avoid indirect leakage observable at the network level. Moreover, while most of the related work aims at preventing traditional progress-insensitive non-interference, we are considering progress-sensitive non-interference, where an attacker can indirectly observe the progress of an object, caused by e.g. process termination or suspension. To the best of our knowledge, there is no prior work considering a concept similar to interaction non-interference, which stipulates indistinguishability of interactions between distributed objects for a network viewer observing the messages exchanged through method calls on the communication channels in the network, given that the communicated low input values are the same.

In general, techniques that come with different goals might also have some similarities. For example, model-based verification has a significant different goal than our work. For instance, the main difference between our work and [41] is that in our case there is no exploitable bug in the program. Instead, we consider legal program behavior that might be informative to attackers. The attack model and assumptions make remarkable differences as well. For example, in our setting, there is no direct interaction with the malicious agent, the interaction is indirect though asynchronous method calls. With respect to non-interference, we are focusing on security leakages where the attacker is capable of observing the interactions between agents, which for example is totally different from works where attackers have interactions with the system, e.g. [42]. However, by our trace analysis, we are also looking at runtime state changes and try to prevent reaching states that may result in information leakage according to interaction non-interference. This has similarities to to model-based verification techniques such as Hoare logic, model checking, or state-based analysis approaches. In contrast to Hoare logic we avoid verification conditions requiring (non-trivial) theorem proving, and in contrast to model checking we transform a program to a trace expression used for further analysis.

9. Conclusion

We have studied non-interference for concurrent distributed object systems communicating by means of asynchronous method calls. The concurrent objects may communicate confidential and non-confidential information, restricting confidential information to method parameters/returns declared as safe channels for confidential information. Due to the non-deterministic nature of such systems and due to the non-trivial implicit information flow leakage related to observation of communication patterns, standard definitions of non-interference are not suitable. We have defined a notion of *interaction non-interference* and have shown how to enforce it by static analysis, using a type and effect system for secrecy levels and using analysis of communication traces addressing indirect network leakage. The analysis is modular and is done class-wise, and we have outlined a proof for soundness. We have considered an object-oriented language centered around the chosen concurrency model. The setting of concurrent objects and object-orientation gives some benefits as well as some challenges, compared to other settings. The benefits include:

- *protection of state.* Each object encapsulates its state in the sense that remote access is not allowed. This means that we do not restrict information flow between confidential and non-confidential variables as long as they do not affect communication behavior to cause a leakage. All fields are private and their secrecy level may vary dynamically with the static knowledge of their values and with the implicit context of high level if- and while-tests.
- *concurrency control.* The high level await mechanism allows cooperative scheduling with explicit control of process suspension and resumption without low level mechanism such as locking or signaling mechanisms. This enables a compositional analysis.
- *message-oriented communication.* The underlying message passing mechanism for method interaction defines one-way as well as two-way interactions. Based on our analysis, the implicit leakage at the network level can then be addressed by expressing communication traces for each method.
- *inflation of high levels.* Our approach does not lead to inflation of high levels, since method calls in a high context do not require methods to be high, and since fields and program variables may go from a high level to a low level.

However, this setting implied some challenges, which we have addressed:

- *secrecy level invariants.* The presence of suspension points imply that secrecy levels of fields may change during suspension. We use an approach similar to class invariants for controlling the level of fields during suspension. Secrecy levels of fields must be maintained upon suspension and method completion. In contrast, the levels of local variables may change freely since their values are not modified during suspension.
- *modularity.* For a given object, the precise timing of observable input events, reflecting method invocation and completion to the object, cannot be detected statically since these events cannot be determined from the program code. This makes the trace analysis less direct. We solve this by considering trace expressions that include reaction events. These are by definition non-observable, but give partial information about the timing of the corresponding input events, which makes the analysis less direct.
- *implicit self-calls.* Self-calls pose non-trivial challenges for the modular analysis, since the static analysis cannot in general detect if a call $o.m(\ldots)$ is a self-call or not, and since a self-call may indirectly have observable effects (when the called method calls external objects). We solve this by including self-calls in the trace expressions, making special considerations for calls detected as self-calls (i.e., calls to this).

The considered language is small, but includes mechanisms for process control, which often is defined by the underlying operating system. With a dedicated virtual machine this makes it possible to limit attacks from within the underlying operating system.

Future Work. As future work, we will consider other language features such as inheritance, which was not considered here, enrich our static analysis, and providing a hybrid approach to satisfy the interactive non-interference policy combining runtime and static analysis. The latter point requires an operational semantics assigning runtime level to objects as well as to values of program variables.

Inheritance and late binding will complicate the analysis in that the binding of a method call is not in general static. As our approach depends on static binding to be able to compute the traces, it cannot be extended to deal with inheritance in a straight forward manner. However, the approach for partial correctness reasoning used in [43, 44] allows modular reasoning for each (sub)class with static binding of self calls, based on the assumption that the runtime class of the considered object is the same as the considered class. Calls other than (explicit) self-calls are statically controlled by interfaces (as in SeCreol). This means that trace sets of inherited methods need to be recalculated in a subclass (due to possible renewed bindings of Self-calls). Using the approach of [43], we may extend the current class-wise static analysis of non-interference to deal with inheritance and late binding.

The concept of cointerface was used in the language to allow type-correct callbacks to external caller objects. This concept gives the possibility of stating minimal security requirements to callers of methods of an interface. We would like to explore this possibility in future work.

To enrich the static analysis, we aim to use the program dependency graphs (PDGs) by considering the effects of program control flow and data flow [45] on interactive communication among objects with security levels which is also proven at least as powerful as security type systems in detecting potential information flows [46] while it can decrease false positives even more in progress-sensitive approaches. In addition, to improve the enforcement, we consider dynamic labeling and decentralized label model (DLM) [36] to provide a hybrid enforcement mechanism as future work.

Acknowledgment

This work was partially supported by the project IoTSec - Security in IoT for Smart Grids, with number 248113/O70 part of the IKTPLUSS program funded by the Norwegian Research Council, and by the project SCOTT (www.scott-project.eu) funded by the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737422. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme of Austria, Spain, Finland, Ireland, Sweden, Germany, Poland, Portugal, Netherlands, Belgium, and Norway.

References

[1] N. Heintze, J. G. Riecke, The SLam calculus: programming with secrecy and integrity, in: In POPL'98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1998, pp. 365–377.

[2] D. Hedin, A. Sabelfeld, A perspective on information-flow control., in: Software Safety and Security - Tools for Analysis and Verification, Vol. 33 of NATO Science for Peace and Security Series - D: Information and Communication Security, IOS Press, 2012, pp. 319–347.

[3] M. R. Clarkson, F. B. Schneider, Hyperproperties, J. Comput. Secur. 18 (6) (2010) 1157–1210.
URL `http://dl.acm.org/citation.cfm?id=1891823.1891830`

[4] J. A. Goguen, J. Meseguer, Unwinding and inference control, in: IEEE Symposium on Security and Privacy, 1984, pp. 75–75.

[5] N. Heintze, J. G. Riecke, The SLam calculus: Programming with secrecy and integrity, in: POPL'98, POPL'98, ACM, 1998, pp. 365–377.

[6] A. Askarov, S. Hunt, A. Sabelfeld, D. Sands, Termination-insensitive noninterference leaks more than just a bit, in: Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security, ESORICS '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 333–348. `doi:10.1007/978-3-540-88313-5_22`.
URL `http://dx.doi.org/10.1007/978-3-540-88313-5_22`

[7] Open web application security project (owasp) top 10 2010 and 2013, `http://www.owasp.org/index.php` (2017).

[8] S. Christey, J. E. Kenderdine, et.al., Common weakness enumeration (cwe version 2.9) (2015).

[9] E. B. Johnsen, O. Owe, An asynchronous communication model for distributed concurrent objects, Software and Systems Modeling 6 (1) (2007) 35–58.

[10] L. Zheng, A. C. Myers, Dynamic security labels and static information flow control, International Journal of Information Security 6 (2) (2007) 67–84. `doi:10.1007/s10207-007-0019-9`.
URL `http://dx.doi.org/10.1007/s10207-007-0019-9`

[11] C. A. R. Hoare, Communicating Sequential Processes, International Series in Computer Science, Prentice Hall, 1985.

[12] M. Broy, K. Stølen, Specification and Development of Interactive Systems, Monographs in Computer Science, Springer, 2001.

[13] O.-J. Dahl, Verifiable Programming, International Series in Computer Science, Prentice Hall, 1992.

[14] O.-J. Dahl, Object-oriented specifications, in: Research directions in object-oriented programming, MIT Press, Cambridge, MA, USA, 1987, pp. 561–576.

[15] E. B. Johnsen, O. Owe, I. C. Yu, Creol: A type-safe object-oriented model for distributed concurrent systems, Theoretical Computer Science 365 (1–2) (2006) 23–66.

[16] U. Erlingsson, The inlined reference monitor approach to security policy enforcement, Ph.D. thesis, Cornell University, Ithaca, NY, USA, aAI3114521 (2004).

[17] T. Kremenek, D. Engler, Z-ranking: Using statistical analysis to counter the impact of static analysis approximations, in: International Static Analysis Symposium, Springer, 2003, pp. 295–315.

[18] O. Owe, T. Ramezanifarkhani, Confidentiality of interactions in concurrent object-oriented systems, in: J. Garcia-Alfaro, G. Navarro-Arribas, H. Hartenstein, J. Herrera-Joancomartí (Eds.), Data Privacy Management, Cryptocurrencies and Blockchain Technology, Springer International Publishing, Cham, 2017, pp. 19–34.

[19] C. C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, J. Log. Algebr. Meth. Program. 83 (5-6) (2014) 360–383. `doi:10.1016/j.jlamp.2014.03.003`.
URL `http://dx.doi.org/10.1016/j.jlamp.2014.03.003`

[20] D. E. Denning, P. J. Denning, Certification of programs for secure information flow, Communications of the ACM 20 (7) (1977) 504–513.

[21] A. C. Myers, Jflow: Practical mostly-static information flow control, in: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1999, pp. 228–241.

[22] D. Volpano, G. Smith, C. Irvine, A sound type system for secure flow analysis, Journal of Computer Security.

[23] V. Simonet, The Flow Caml system. Software release, available at `cristal.inria.fr/~simonet/flowcaml` (Jul. 2003).

[24] V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, R. Sekar, Provably correct runtime enforcement of non-interference properties, in: International Conference on Information and Communications Security, Springer, 2006, pp. 332–351.

[25] K. R. M. Leino, R. Joshi, A semantic approach to secure information flow, Lecture notes in computer science 1422 (1998) 254–271.

[26] Á. Darvas, R. Hähnle, D. Sands, A theorem proving approach to analysis of secure information flow, in: International Conference on Security in Pervasive Computing, Springer, 2005, pp. 193–209.

[27] T. H. Austin, C. Flanagan, Efficient purely-dynamic information flow analysis, ACM Sigplan Notices 44 (8) (2009) 20–31.

[28] D. Devriese, F. Piessens, Noninterference through secure multi-execution, in: Security and Privacy (SP), 2010 IEEE Symposium on, IEEE, 2010, pp. 109–124.

[29] L. Zheng, A. C. Myers, Dynamic security labels and noninterference, in: Formal Aspects in Security and Trust, Springer, 2005, pp. 27–40.

[30] S. Tse, S. Zdancewic, Run-time principals in information-flow type systems, ACM Transactions on Programming Languages and Systems (TOPLAS) 30 (1) (2007) 6.

[31] A. Sabelfeld, A. Russo, From dynamic to static and back: Riding the roller coaster of information control research, in: E. Clarke, I. Virbitskaite, A. Voronkov (Eds.), 8th International Andrei Ershov Memorial Conference, PSI 2011, Novosibirsk, Russia, June 27 - July 1, 2011, Revised Selected Papers, Vol. 7162 of *Lecture Notes in Computer Science*, Springer, 2012.

[32] A. Russo, A. Sabelfeld, Dynamic vs. static flow-sensitive security analysis, in: Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010, IEEE Computer Society, 2010, pp. 186–199.

[33] L. Beringer, End-to-end multilevel hybrid information flow control, in: Asian Symposium on Programming Languages and Systems, Springer, 2012, pp. 50–65.

[34] P. Buiras, D. Vytiniotis, A. Russo, HLIO: mixing static and dynamic typing for information-flow control in haskell, in: K. Fisher, J. H. Reppy (Eds.), Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015, ACM, 2015, pp. 289–301. doi:10.1145/2784731.2784758.
URL http://doi.acm.org/10.1145/2784731.2784758

[35] B. Sayed, Protection against malicious javascript using hybrid flow-sensitive information flow monitoring, Ph.D. thesis, Department of Electrical andC omputer Engineering, University of Victoria, Canada (2015).

[36] J. Liu, M. George, K. Vikram, X. Qi, L. Waye, A. Myers, Fabric: A platform for secure distributed computation and storage, 2009, pp. 321–334.

[37] G. L. Guernic, Automata-based confidentiality monitoring of concurrent programs, in: Proceedings of IEEE Computer Security Foundations Symposium, 2007, pp. 218–232.

[38] M. S. Miller, Robust composition: Towards a unified approach to access control and concurrency control, Ph.D. thesis, John Hopkins University (2006).

[39] C. Hammer, G. Snelting, Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs, International Journal of Information Security 8 (6) (2009) 399–422.

[40] C. Hammer, Experiences with pdg-based ifc, in: International Symposium on Engineering Secure Software and Systems, Springer, 2010, pp. 44–60.

[41] K. Fisher, J. Launchbury, R. Richards, The hacms program: using formal methods to eliminate exploitable bugs, Phil. Trans. R. Soc. A 375 (2104) (2017) 20150401.

[42] D. C. Wardell, R. F. Mills, G. L. Peterson, M. E. Oxley, A method for revealing and addressing security vulnerabilities in cyber-physical systems by modeling malicious agent interactions with formal verification, Procedia computer science 95 (2016) 24–31.

[43] O. Owe, Verifiable programming of object-oriented and distributed systems, in: L. Petre, E. Sekerinski (Eds.), From Action Systems to Distributed Systems - The Refinement Approach., Chapman and Hall/CRC, 2016, pp. 61–79. doi:10.1201/b20053-8.
URL http://dx.doi.org/10.1201/b20053-8

[44] O. Owe, Reasoning about inheritance and unrestricted reuse in object-oriented concurrent systems, in: E. Ábrahám, M. Huisman (Eds.), Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings, Vol. 9681 of Lecture Notes in Computer Science, Springer, 2016, pp. 210–225. doi:10.1007/978-3-319-33693-0_14.
URL http://dx.doi.org/10.1007/978-3-319-33693-0_14

[45] T. Ramezanifarkhani, M. Razzazi, Principles of data flow integrity: Specification and enforcement., J. Inf. Sci. Eng. 31 (2) (2015) 529–546.

[46] H. Mantel, H. Sudbrock, Types vs. pdgs in information flow analysis, in: International Symposium on Logic-Based Program Synthesis and Transformation, Springer, 2012, pp. 106–121.

[47] C. C. Din, J. Dovland, E. B. Johnsen, O. Owe, Observable behavior of distributed systems: Component reasoning for concurrent objects, *Journal of Logic and Algebraic Programming* 81 (3) (2012) 227–256.

Appendices

A. Operational Semantics

We here present the operational semantics of the core language. The main purpose of this semantics is to understand the communication traces appearing at runtime. We therefore omit the complication of assigning runtime secrecy levels to objects and values of program variables. An operational semantics with secrecy levels is presented in [18]. The semantics formalizes the notion of process queue (PQ), idleness, and generation of events (as labels on the transition relation). Thus a sequence of execution steps gives rise to a sequence of events, capturing the history. Generation of identities for objects and method calls is handled by underlying semantics functions and implicit attributes. The operational semantics uses an additional construct **get** to deal with (the completion of) call statements, letting **get** u appear as a right-hand-side, where u denotes a method call identity. The *query* v:=get u will block while waiting for completion of u and v:=await get u will suspend. We use the notation explained above for mappings, and a denotes an object expression, b denotes a Boolean expression, o denotes an object identity, u denotes a method identity, and d denotes a value (a data value or an object identity).

For simplicity we omit rules for while. While can be handled by expanding a while-statement to be executed to an if-statement with an inner while upon execution of the the while-statement. The semantics of the while-statement while b do s od is equivalent to that of if b then s; while b do s od fi. The semantics of an if-statement without else-part, if b then s; while b do s od fi, is equivalent to if b then s else skip fi.

The operational semantics of the core language is given in Fig. 18. A runtime configuration of a system is seen as a multiset of objects and messages (using blank-space as a binary multiset constructor). Each rule in the operational semantics deals with only one object o, and possibly messages, reflecting that we deal with concurrent distributed systems communicating asynchronously. When a subconfiguration c can be rewritten to a c', this means that the whole configuration $\dots c \dots$ can be rewritten to $\dots c' \dots$, reflecting interleaving semantics. Each object o is responsible for executing all method calls to o as well as self-calls. An object has at most one active process, reflecting a method execution, and a sequence of suspended processes organized in a process queue PQ. Remote calls and replies are handled by messages. Objects have the form

$$o : \mathbf{ob}(\delta, \overline{s})$$

where o is the object identity, δ is the current object state, and $\overline{s}$ is a sequence of statements ending with a **return**, representing the remaining part of the active process, or **idle** when no active process. A message have the form

$$\texttt{msg}\ o \to o'.m(\overline{e})$$

representing a call event, where o is caller, o' callee, m the method and $\overline{e}$ the actual parameter values, or

$$\texttt{msg}\ o \leftarrow o'.(u, d)$$

representing a completion event where d is the returned value and u the identity of the call.

In the operational semantics rules, pc is the confidentiality level of the object that is going to execute an instruction at the current program point. Moreover, the operational semantics uses some additional variables, like PQ for holding the process queue and nextId and nextOb for generating unique identities for calls and objects. These appear as fields in the operational semantics. Furthermore, this is handled as an implicit class parameter, while callId and caller appear as implicit method parameters, holding the identity of a call and its caller, respectively. The operational semantics uses an additional *query* statement, [**await**] **get** u, for dealing with the termination of call/await call statements. The syntax [**await**] denotes an optional **await**. The query **get** u is blocking while waiting for the method response with identity u, and **await get** u is a suspending query.

The state of an object is given by a twin mapping, written $(\alpha|\beta)$, where α is the state of the field variables (including PQ, nextId, nextOb) and class parameters $\overline{cp}$ (including this), and β is the state of the local variables and formal parameters (including callId and caller) of the current process. Look-up in a twin mapping, $(\alpha|\beta)[z]$, is simply given by $(\alpha+\beta)[z]$. For an expression e, the notation $\alpha[z := e]$ abbreviates $\alpha[z \mapsto$

$alpha[e]]$, and the notation $(\alpha|\beta)[v := e]$ abbreviates **if** v *in* β **then** $(\alpha \,|\, \beta[v \mapsto (\alpha|\beta)[e]])$ **else** $(\alpha[v \mapsto (\alpha|\beta)[e]] \,|\, \beta)$, where *in* is used for testing domain membership.

The *process queue* PQ is the queue of suspended processes, of form $(\beta, \overline{s}))$. The operations $enq(PQ, p)$ and $deq(PQ, \alpha)$ are used to add a process p to the queue, and to select an *enabled* process (if any) from the queue, respectively. The latter results in the sequence $(p; PQ')$ of the selected enabled process p and the remainder of the queue PQ' (depending on the specific scheduling policy), or the empty sequence *empty* if no process is enabled. A process $(\beta, \overline{s})$ is *enabled* if it starts with an enabled statement. A conditional **await** statement is enabled if the condition evaluates to true (in state $\alpha|\beta$), and an **await** call statement is not enabled (unless reduced by the QUERY rule). All other statements are enabled.

Asynchronous (simple) method invocation is captured by the rule SIMPLE CALL/CALL. The generated call identity is locally unique (and globally unique in combination with the parent object). The call identity generated by this rule is passed through an invocation message, which is to be consumed by the callee object by the rule START. When an object has no active process, denoted ***idle***, a suspended process may be continued (by rule CONTINUE), given that the process is enabled, or a method call is selected for execution by rule START. The invocation message is removed from the configuration by this rule, and the identity of the call is assigned to the implicit parameter callId. With rule RETURN, a return value is generated upon method termination and passed in a completion message together with the call identity stored in callId. The return value is fetched by rule QUERY. Note that a query statement blocks until the corresponding future value is generated by rule RETURN.

The QUERY rule says that an occurrence of **await** $v :=$ **get** u, or $v :=$ **get** u, in object o is replaced by the assignment $v := d$ when the completion **msg** $o \leftarrow o'.(u, d)$ appears. The keyword **await** is removed when in front of such a query statement. Note that rule QUERY removes the completion message from the configuration, which is possible since any corresponding **get** will be found in the object when the completion message appears There is at most one such occurrence (in the first statement of either the active statement list or a process in PQ). If object o does not contain **get** u then the completion message is removed without any effect on o. This happens when the corresponding call was a simple call. In Rule START, we assume that m is bound to a method with local state β (including default values) and code $\overline{s}$. Note that bindings for the parameters $\overline{y}$ and the implicit parameter nextId are added to the local state.

Object creation is captured by the rule NEW. The generated object identity is locally unique, and also globally unique since the object identity is given by a generator term embedding the parent object. The generated object gets this identity. Here $init_C$ denotes the initialization statements (the constructor) of class C, and δ_C denotes the initial state of class C with default/initial values for the fields. The binding of class parameters and this is added explicitly (by $\mathsf{this} \mapsto \delta[\mathsf{nextOb}]$ and $\overline{cp} \mapsto \delta[\overline{e}]$). We obtain an active object by letting *init* initiate internal activity, using simple self-calls to allow the object to interleave continued internal activity with reaction to external calls. The initialization statements of a program will typically create the other initial objects.

In the case that an await statement is not enabled, the current process is placed on the process queue and the object becomes *idle*, as described by rule SUSPEND. An idle object may next start a new process (according to rule START) or continue with an enabled process from the process queue (according to rule CONTINUE). This choice depends on the underlying scheduling inside an object. The given language fragment may be extended with constructs for local (stack-based) method calls, e.g., by using the approach of [47]. As we focus on inter-object communication, this is omitted here. For simplicity we omit runtime secrecy levels and therefore the result of evaluating a secrecy comparison (by $\sqsubseteq$) is not defined in the operational semantics.

For a given program (and starting object) the operational semantics defines a set of *executions*, each given by a sequence of global states (configurations). The state of an execution E at time t is the state given by $E[t]$. A sequence of execution steps $E[i] \xrightarrow{e_i} E[i+1] \xrightarrow{e_{i+1}} E[i+2] \xrightarrow{e_{i+2}} \ldots$ generates the trace $e_i; e_{i+1}; e_{i+2} \ldots$. Even if an execution E may be infinite, our analysis will deal with finite segments. In our concurrency model the objects compute independently at their own speed (when not blocked), and we assume that one object is not unboundedly delayed (unless blocked). Thus for our concurrency model we may assume *inter-object fairness.*

SKIP:

$$\begin{array}{rl} & o : \mathbf{ob}(\delta, \mathtt{\mathbf{skip}}; \overline{s}) \\ \xrightarrow{empty} & o : \mathbf{ob}(\delta, \overline{s}) \end{array}$$

ASSIGN :

$$\begin{array}{rl} & o : \mathbf{ob}(\delta, v := e; \overline{s}) \\ \xrightarrow{empty} & o : \mathbf{ob}(\delta[v := e], \overline{s}) \end{array}$$

IF-TRUE :

$$\begin{array}{rl} & o : \mathbf{ob}(\delta, \mathtt{\mathbf{if}}\ b\ \mathtt{\mathbf{then}}\ \overline{s1}\ \mathtt{\mathbf{else}}\ \overline{s2}\ \mathtt{\mathbf{fi}}; \overline{s}) \\ \xrightarrow{empty} & o : \mathbf{ob}(\delta, \overline{s1}; \overline{s}) \\ & \mathtt{\mathbf{if}}\ \delta[b] = true \end{array}$$

IF-FALSE :

$$\begin{array}{rl} & o : \mathbf{ob}(\delta, \mathtt{\mathbf{if}}\ b\ \mathtt{\mathbf{then}}\ \overline{s1}\ \mathtt{\mathbf{else}}\ \overline{s2}\ \mathtt{\mathbf{fi}}; \overline{s}) \\ \xrightarrow{empty} & o : \mathbf{ob}(\delta, \overline{s2}; \overline{s}) \\ & \mathtt{\mathbf{if}}\ \delta[b] = false \end{array}$$

NEW :

$$\begin{array}{rl} & o : \mathbf{ob}(\delta, v := \mathtt{\mathbf{new}}\ C(\overline{e}); \overline{s}) \\ \xrightarrow{o\ \leftrightarrow\ \delta[\mathsf{nextOb}].C(\delta[\overline{e}])} & o : \mathbf{ob}(\delta[v := \mathsf{nextOb}, \mathsf{nextOb} := next(\mathsf{nextOb})], \overline{s}) \\ & \delta[\mathsf{nextOb}] : \mathbf{ob}(\delta_C[\mathsf{this} \mapsto \delta[\mathsf{nextOb}], \overline{cp} \mapsto \delta[\overline{e}]], init_C) \end{array}$$

SIMPLE CALL :

$$\begin{array}{rl} & o : \mathbf{ob}(\delta, a!m(\overline{e}); \overline{s}) \\ \xrightarrow{o \rightarrow \delta[a].m(\delta[\mathsf{nextId}, \overline{e}])} & o : \mathbf{ob}(\delta[\mathsf{nextId} := next(\mathsf{nextId})], \overline{s}) \\ & \mathtt{\mathbf{msg}}\ o \rightarrow \delta[a].m(\delta[\mathsf{nextId}, \overline{e}]) \end{array}$$

CALL :

$$\begin{array}{rl} & o : \mathbf{ob}(\delta, [\mathtt{\mathbf{await}}]\ v := a.m(\overline{e}); \overline{s}) \\ \xrightarrow{o \rightarrow \delta[a].m(\delta[\mathsf{nextId}, \overline{e}])} & o : \mathbf{ob}(\delta, [\mathtt{\mathbf{await}}]\ v := \mathtt{\mathbf{get}}\ \delta[\mathsf{nextId}]; \overline{s}) \\ & \mathtt{\mathbf{msg}}\ o \rightarrow \delta[a].m(\delta[\mathsf{nextId}, \overline{e}]) \end{array}$$

START :

$$\begin{array}{rl} & \mathtt{\mathbf{msg}}\ o' \rightarrow o.m(u, \overline{d}) \\ & o : \mathbf{ob}((\alpha|\beta'), \mathbf{idle}) \\ \xrightarrow{o' \twoheadrightarrow o.m(u, \overline{d})} & o : \mathbf{ob}((\alpha|(\beta[\mathsf{caller} \mapsto o', \mathsf{callId} \mapsto u, \overline{y} \mapsto \overline{d}])), \overline{s}) \\ & \mathtt{\mathbf{where}}\ m \text{ is statically bound to } (m, \overline{y}, \beta, \overline{s}) \text{ in the class of } \mathsf{this} \end{array}$$

RETURN :

$$\begin{array}{rl} & o : \mathbf{ob}(\delta, \mathtt{\mathbf{return}}\ e) \\ \xrightarrow{\delta[\mathsf{caller}] \leftarrow \delta[\mathsf{this}].(\delta[\mathsf{callId}], \delta[e])} & o : \mathbf{ob}(\delta, \mathbf{idle}) \\ & \mathtt{\mathbf{msg}}\ \delta[\mathsf{caller}] \leftarrow \delta[\mathsf{this}].(\delta[\mathsf{callId}], \delta[e]) \end{array}$$

QUERY :

$$\begin{array}{rl} & \mathtt{\mathbf{msg}}\ o \leftarrow o'.(u, d) \\ & o : \mathbf{ob}(\ldots [\mathtt{\mathbf{await}}]\ v := \mathtt{\mathbf{get}}\ u\ \ldots) \\ \xrightarrow{o \twoheadleftarrow o'.(u, d)} & o : \mathbf{ob}(\ldots v := d \ldots) \end{array}$$

AWAIT :

$$\begin{array}{rl} & o : \mathbf{ob}(\delta, \mathtt{\mathbf{await}}\ b; \overline{s}) \\ \xrightarrow{empty} & o : \mathbf{ob}(\delta, \overline{s}) \\ & \mathtt{\mathbf{if}}\ \delta[b] = true \end{array}$$

CONTINUE :

$$\begin{array}{rl} & o : \mathbf{ob}((\alpha|\beta'), \mathbf{idle}) \\ \xrightarrow{empty} & o : \mathbf{ob}((\alpha[PQ \mapsto rest])|\beta), \overline{s}) \\ & \mathtt{\mathbf{if}}\ deq(\alpha[PQ], \alpha) = ((\beta, \overline{s}); rest) \end{array}$$

SUSPEND :

$$\begin{array}{rl} & o : \mathbf{ob}((\alpha|\beta), \overline{s}) \\ \xrightarrow{empty} & o : \mathbf{ob}((\alpha[PQ \mapsto enq(\alpha[PQ], (\beta, \overline{s}))], \varepsilon), \mathbf{idle}) \\ & \mathtt{\mathbf{if}}\ \overline{s} \text{ starts with } \mathtt{\mathbf{await}} \end{array}$$

Figure 18: Operational rules reflecting small-step semantics of SeCreol (without secrecy levels).