

On Detecting Over-Eager Concurrency in Asynchronously Communicating Concurrent Object Systems

Olaf Owe^a, Charlie McDowell^b,

^a *University of Oslo, Norway*

^b *University of California, Santa Cruz, USA*

Abstract

Modern software systems are often distributed, and object-orientation is a leading paradigm for system modeling and design. We consider an object-oriented concurrency model for distributed systems, based on active objects, asynchronous method calls, and shared futures. This approach is appealing in that it gives rise to massive parallelism while avoiding active waiting and explicit locks, and has a simple semantics.

In this paper we show that systems developed using active objects and asynchronous method calls can result in system failure due to over-eager concurrency, which we call flooding. A system may feed an object with more calls than it is able to handle, in some cases even regardless of its processing speed. We refer to this situation as *flooding* of the object. We distinguish between *strong* and *weak flooding*. In particular, the notion of strong flooding could lead to problems such as non-responsive objects, system crash, overfull buffers or massive amounts of lost messages, even in the presence of fair scheduling. We present an algorithm to statically detect strong and weak flooding, and prove the soundness of the algorithm.

Keywords: active objects, concurrent objects, asynchronous communication, futures, static analysis, concurrency handling, object flooding, scheduling

1. Introduction

Distributed Systems play an import role in modern society. Conceptually, a distributed system consists of a number of autonomous components that are connected by a network and are cooperating by means of message passing [5]. Thus concurrency is a key aspect of such systems. Components that run in parallel on different computers or on the same multi-threaded computer, may achieve efficient overall performance, provided they are not blocking each other. Modularity of the components is desirable, as this helps in understanding and

[☆]This work was done in the context of the *IoTSec* project funded by the Norwegian Research Council (<http://cwi.unik.no/wiki/IoTSec:Home>). A research exchange period at UCSC 2014/2015 was supported by University of Oslo.

Email addresses: olaf@ifi.uio.no (Olaf Owe), mcdowell@ucsc.edu (Charlie McDowell)

designing complex systems, and thereby enhances scalability. In particular synchronization mechanisms should be modular, avoiding inter-component signaling/notification.

The Actor model [16, 2, 1] has been acknowledged as a natural way of programming distributed and concurrent systems, and is based on a compositional semantics, as opposed to the thread-based concurrency model. It unifies concurrency with the view of actors as autonomous units communicating asynchronously through messages. The Actor model has been adapted to the object-oriented setting in the form of active concurrent objects, interacting by means of remote method calls. The notion of so-called asynchronous methods are implemented by asynchronous message passing, while *suspension* may be used to achieve non-blocking calls [20]. Method interaction inherently supports the notion of request and reply, which provides more control than pure message passing languages, and is therefore attractive for modeling of distributed systems [2, 4].

Shared futures enable even more efficient interaction, allowing objects to share computation results without waiting for the results [31, 14, 25]. A future is a reference to a globally accessible location where the result of a given method call will be stored (once computed), generated by the caller object when the call is made. By allowing futures to be passed as parameters, and thereby shared, several objects may get access to the same future, reading the result held in the future or checking if such a result has appeared. A caller that does not need the result of a called method may pass the associated future to other objects without waiting for the result to appear. Futures are used in several languages [28, 8, 4].

A core language based on this concurrency model is given in Sec. 2. Inter-object concurrency comes for free in the sense that each object can run concurrently with other objects. Synchronization is handled in a modular manner without the use of external notification, by means of conditional suspension. This gives a high degree of autonomy and concurrency, as well as scalability; and high performance is possible due to asynchronous method calls, suspension, and shared futures.

However, this unrestricted concurrency model comes with a price. This programming style may give rise to deadlocks, and it is easy to create programs that are class-wise semantically correct but that fail due to over-eager creation of method calls. A system may feed an object with more calls than it is able to handle, in some cases even regardless of its processing speed. We refer to this situation as *flooding* of the object. Flooding is caused by execution cycles that give rise to unbounded amounts of uncompleted method calls, as discussed further in Sec. 3. Flooding may lead to non-responsive objects, system crash, overfull buffers or massive amounts of lost messages. Thus, flooding may seriously affect the performance of the whole system.

Our work is motivated by experimentation with this concurrency model using an implementation in Java, given as a Java library package hiding the underlying Java thread-based concurrency model [26]. We experienced executions where certain objects were not able to do anything due to an unbounded number of incoming requests, and were surprised by discovering several cases of this kind of flooding, resulting from small changes to sample programs, see illustrations in Sec. 3. This was not expected since the implementation was fair with respect to the underlying threads. Our investigation includes an understanding of the issue of flooding, its seriousness, and a way to detect cases of flooding.

In this paper we define and exemplify the concept of flooding. As flooding may depend on the underlying scheduling, we distinguish between strong and weak flooding. Strong flooding is defined as flooding under “favorable” scheduling, and weak flooding is flooding under weak scheduling assumptions. Thus strong flooding represents the more critical flooding situations. The scientific contribution of the paper is to present a static analysis method to detect possible weak and strong flooding (Sec. 4) by means of a terminating algorithm, and prove its soundness (no false negatives, Sec. 5). Static analysis of flooding cannot be both sound and complete (and terminating), since that would imply static control of termination of cycles depending on Boolean conditions. Due to over-approximation, detection of flooding may not imply a real flooding situation. However, when no flooding is detected, this implies that there is no real flooding situation (soundness). We discuss improvements for tighter detection of flooding, and in particular of strong flooding (Sec. 6) based on assumptions on the underlying process scheduling.

Related work. Fairness of the underlying process scheduling inside an object, as well as load balancing between objects, may eliminate weak flooding. Thus the topics of process scheduling and load balancing are complementary to the current work. Detection of flooding naturally relies on underlying assumptions about these factors. Detection of weak flooding is valuable for systems with weak underlying scheduling conditions, while strong flooding is valuable for systems with favorable scheduling conditions. Also deadlock analysis is complementary, since a dead object may flood in a trivial sense since it cannot complete any incoming calls. Our results on strong flooding will rely on assumptions about fair scheduling and absence of deadlock.

While analysis of deadlock situations, scheduling, and load balancing for this concurrency model have been investigated in several ways [7, 15, 17, 22], we are not aware of analysis of object flooding for this or similar concurrency models. In particular static analysis of the suspension, non-blocking call, and future mechanisms are challenging. These mechanisms are essential for efficient high-level programming of concurrent object systems, and investigations of these are therefore valuable.

Arvind and Nikhil [3] recognized a problem of “excessive parallelism” in the context of the functional data flow language Id and tagged-token data flow. More recently, there have been efforts to address scheduling and fairness issues with active objects, but none of that work discusses the issue of system failure due to flooding. Instead, scheduling has been proposed to improve performance, and in some cases as an essential part of the correctness of the algorithm [27].

The fundamental work of Ganty and Majumdar [12] considers asynchronous programs and clarifies the complexity of static verification of safety and liveness properties, using a translation to Petri Nets. However, their asynchronous programs are different from the ones considered here, in that blocking is avoided by passing control from one unit to another, instead of using unit-local suspension as in our model of asynchronously communicating active objects. Another difference is that our study provides insight in the considered concurrency model, rather than reducing the problems to reachability questions embedded in some other formalism. In particular, the connection between weak and strong flooding and the notions of underlying scheduling and fairness relate to the considered concurrency model.

In thread-based systems, programs may generate an unbounded number of threads, for instance as a result of incoming requests. This problem is considered in [30], and a static algorithm for detecting unbounded thread-instantiation loops is presented. The approach is heuristic and is not guaranteed to find all unbounded loops, and it may report unbounded loops that are actually bounded. The analysis is based on analysis of execution loops, similar to our approach. However, our analysis in the setting of execution cycles (possibly across several objects) involving suspension while waiting for futures, is technically quite different from their setting of loops with object creation. And our approach aims at a soundness result, rather than heuristics.

Another difference is that in [30], the detection of termination was central. Termination of execution cycles is clearly related to flooding, since a terminating cycle cannot create an unbounded number of calls (nor an unbounded number of objects). As loops traversing through finite data structures typically terminate, detection of such cases was essential in [30]. In our framework, local data structures are defined using a functional sublanguage for data types defined by generators, and (recursive) functions over such structures are defined by means of *terminating generator induction* (TGI) [6], which may be checked statically. Thus functional expressions terminate in our setting. For instance, we may notify selected members of a list l , by means of a single statement ($select(l)!m(...)$), by making a multicast to the result of $select(l)$ where $select$ is a TGI-defined function traversing a list and returning a sublist. Thus such a statement is known to terminate. In our setting, execution cycles at the imperative level often represents infinite behavior of non-terminating systems, therefore we do not include heuristics for termination detection. Such detection could well be added to our framework, for instance the main example shows a terminating imperative execution cycle (as well as a multi-cast).

Static analysis of unbounded object structures was also considered by Grabe [13]. In particular, the abstract control flow of method calls and of multi-threaded reentrant programs was used for deadlock detection. The static control flow structures needed in the analysis of deadlock are similar to our control flow structures in that they involve several classes, giving a static approximation of the behavior of several objects. However, the analysis itself is quite different. For deadlock detection, waiting for a future or condition can be critical, while for flooding detection, cycles with sufficient waiting conditions do not cause flooding.

Outline. The next section gives an explanation of the considered concurrency paradigm defining a core language. A motivating example is given in Sec. 3, and Sec. 4 defines flooding and related notions as well as an algorithm for detecting flooding reusing the main example. Sec. 5 gives the theoretical results including soundness. Extensions and a discussion of strong flooding are discussed in Sec. 6 Finally, Sec. 7 concludes the paper. An appendix defines the operational semantics of the considered core language.

2. A Core Language

We introduce a minimal language based on the concurrency model motivated above. The language is a variation of Creol [21, 20, 19], supporting concurrent,

In	$::=$	interface I [extends I^+] $\{S^*\}$	interface declaration
Cl	$::=$	class $C([T\ cp]^*)$ [implements I^+] $\{[T\ w\ :=\ e]^*\ [s]\ M^*\}$	class definition
M	$::=$	$S\ B$	method definition
S	$::=$	$T\ m([T\ x]^*)$	method signature
B	$::=$	$\{\textbf{var}\ [T\ x\ :=\ e]^*; [s]; \textbf{put}\ e\}$	method blocks
T	$::=$	$I \mid \text{Int} \mid \text{Bool} \mid \text{String} \mid \text{Void} \mid \text{Fut}[T]$	types
v	$::=$	$x \mid w$	variables (local or field)
e	$::=$	$\text{null} \mid \text{void} \mid \text{this} \mid \text{myfuture} \mid v \mid cp \mid f(\bar{e})$	pure expressions
s	$::=$	$v := e \mid [v :=] \textbf{new}\ C(\bar{e})$ $[v :=] v!m(\bar{e}) \mid [v :=][v.]m(\bar{e})$ $\mid [\textbf{await}][v :=] \textbf{get}\ e \mid \textbf{await}\ e$ $\mid \textbf{if}\ e\ \textbf{then}\ s\ [\textbf{else}\ s]\ \textbf{fi}\ \mid \textbf{skip} \mid s; s$	assignm./object creation call statements suspension / get future basic statements

Figure 1: Core language syntax.

object-oriented, active objects using asynchronous method calls and shared futures as the (only) interaction mechanisms. Unnecessary waiting can be avoided by means of non-blocking method calls, supported by a suspension mechanism, which allows an object to perform other tasks while waiting for a Boolean condition to become true or for a method result to appear. Thus shared variables as well as thread-based notification are avoided. The resulting paradigm has a compositional semantics and supports modular reasoning and understanding of classes, and thus an object can be tested in isolation since its semantics are not changed by its environment [23, 9, 10, 11].

The core language is presented in Fig. 1 using BNF notation extended with the meta-notation $[]^*$, $[]^+$ and $[]$ for repeated, repeated at least once, and optional parts, respectively. The language includes standard statements for assignment, **skip**, conditionals, and sequential composition. We use a Java-like syntax, but use $:=$ for assignments. We let C denote a class name, I an interface name, m a method name, cp a formal class parameter, w a field, x a method parameter or local variable. Expressions e and functions f are terminating and side-effect free, and $\bar{e}$ is a (possibly empty) expression list, comma-separated.

Methods are organized in interfaces and classes in a standard manner. A class C takes a list of formal parameters $\overline{cp}$, and defines fields $\overline{w}$, optional initialization statements $\overline{s}$, and methods $\overline{M}$. There is read-only access to **this**, referring to the current object, as well as the implicit method parameter **myfuture**, referring to the future of the call of the current method. A method definition has the form $m(\overline{x})\{\textbf{var}\ \overline{y};\ s;\ \textbf{put}\ e\}$, when ignoring type information, where $\overline{x}$ is the list of formal parameters, $\overline{y}$ is an optional list of *method-local variables*, s is a sequence of statements, and the final *put* statement writes the value of e in the future of the call, “resolving the future”. A future variable u declared by $\text{Fut}[T]\ u$ may refer to futures containing values of type T . The asynchronous call statement $u := x!m(\bar{e})$ invokes the method m on object x with input values $\bar{e}$. The identity of the generated future is assigned to u , and the calling process (i.e., method execution) continues execution without waiting for u to become resolved. The query statement $v := \textbf{get}\ u$ is used to fetch the future value. This statement blocks until u is resolved, and then assigns the value contained

in u to v . A future variable may be passed as a parameter, allowing objects to share futures.

The non-blocking release statement **await** $v := \mathbf{get} \ u$ suspends the current process as long as u is not resolved, allowing other (enabled) processes of the object to continue. This gives rise to more efficient programming with futures. Similarly, the statement **await** c suspends the current process as long as the Boolean condition c is not satisfied. The remote call $v := x.m(\bar{e})$ blocks while waiting for the future of the call to be resolved, and is a shorthand for $u := x!m(\bar{e}); v := \mathbf{get} \ u$, for a fresh u . The construct for local calls, $v := m(\bar{e})$, employs a standard stack-based execution. Trivial call statements, $x!m(\bar{e})$, are used when the future/result is not needed, and in this case x may be a list of objects. The statement then represents a multi-cast to each object in the list. Object variables are typed by interfaces, and remote field access is (syntactically) forbidden. We assume that call and query statements are well-typed.

As an example, the statements `val := get u; client!push_val(val)` where u is a future, will imply waiting for the future to be resolved, whereas the code `client!push_fut(u)` will send the future to `client` without waiting. (Similar code is found in the variations of the main example.)

The language is strongly typed, and the type system is similar to those of Creol and ABS [24, 18], but is omitted here since we consider well-typed programs. Local data structures are defined by a functional sublanguage for data types giving rise to immutable values (as opposed to concurrent objects which are mutable). Mutable values are passed “by value” and objects are passed “by reference”. We assume a predefined data type for list; for instance, `List[I] l := Nil` declares a list l of objects of interface I initialized to the empty list *Nil*. The assignment `l := append(l, x)` adds an element x to the end of the (immutable) list l . Functions defined in the functional sublanguage are assumed to be terminating, which can be realized by the restricted style of terminating generator induction [6].

In our concurrency model, each object is responsible for executing methods called on that object, and has its own (possibly virtual) CPU. Each method instance corresponds to a *process* at run time, which may be suspended by *await* statements. A started process continues until the end (given by a *put*) or to a suspension point. Suspended processes are kept in a process queue together with incoming calls. An object can only execute one process at a time. When a process is suspended, the object may continue with another enabled process. Local asynchronous calls are queued together with incoming calls, while local synchronous calls are performed as usual, in a stack-based manner. The suspension mechanism allows an object to combine active behavior, typically initiated by the initialization statements, and reactive behavior, initiated by incoming calls. Delays in execution are caused by delayed start of a method execution, say when the processing object is busy, by suspension, and by blocking *gets*. The details of the semantics are explained in the appendix, which defines the operational semantics of the core language.

3. Flooding Cycles

As a motivating example we will consider versions of the publish/subscribe example. Clients may subscribe to a *service* object and the service object

```

1
2 data News = (String content, Int date) // a product data type
3
4 interface ServiceI{
5     Void subscribe(ClientI cl) // called by Clients
6     Void produce()} // called by Proxies
7
8 interface ProxyI{
9     ProxyI add(ClientI cl) // called by Service
10    Void publish(Fut[News]fut)} // called by Service
11
12 interface ProducerI{
13     News detectNews()} // called by Service
14
15 interface NewsProducerI{
16     Void add(News ns) // called when news arrives
17     News getNews()} // called by Producers
18
19 interface ClientI{
20     Void signal(News ns)} // called by Proxies

```

Figure 2: The interfaces of the units in the subscription example.

will ensure that subscribing objects receive information about “news”, here simply defined as a product type consisting of a *content* and a *date*. Clients are notified of news by the `signal` method. The service object is using a number of proxies to handle all of the clients and is using an underlying news producer to obtain news. The service object is using futures to avoid being delayed by waiting for news to be available, thus it may continuously respond to clients. The interfaces are given in Fig. 2. A high-level implementation of the publish/subscribe model is given in Fig. 3, adapted from Din and Owe [11]. For simplicity, **put** void statements are omitted in the code. Note that the statement `myClients!signal(ns)`, in line 28 of Fig. 3, represents a multi-cast to each object in the `myClients` list.

Modifying `Client` and `Proxy` as shown in Fig. 4, results in a program that will flood the system with suspended calls. The change is to shift requiring the actual news to have arrived, from the `Proxy` (as represented by the statements `ns:=get fut; myClients!signal(ns)` in the original *publish* method) to the `Client` (i.e., `news:=get fut` in the modified `Client.signal` method).

This seemingly minor change, and one that would even seem to make sense in the interest of maximizing concurrency, is in fact “too much.” We might naively take it even one step further and have the client instead do `news:=await(fut)`, which has the additional advantage of allowing the `Client` to process the news items as they become available, rather than in the order that the futures were received. In either case, the following sequence of calls can occur, which constitute a flooding cycle.

`Service.produce` asynchronously calls `Producer.detectNews`, line 12 of Fig. 3

```

1 // Monitor news from a NewsProducer and
2 // pass the news on to clients via Proxies.
3 class Service(Int limit, NewsProducerI np)
4     implements ServiceI{
5     ProducerI prod; ProxyI proxy; ProxyI lastProxy;
6     { prod := new Producer(np);
7       proxy:= new Proxy(limit,this);
8       lastProxy:=proxy; this!produce() }
9     Void subscribe(ClientI cl)
10        {lastProxy:=lastProxy.add(cl)}
11     Void produce(){var Fut[News]fut;
12        fut :=prod!detectNews();
13        proxy!publish(fut)}} // sends future, no waiting
14 // Publish news to limited number of clients.
15 // If necessary create additional Proxies.
16 class Proxy(Int limit,ServiceI s) implements ProxyI{
17     List[ClientI] myClients:=Nil; ProxyI nextProxy;
18     ProxyI add(ClientI cl){
19         var ProxyI lastProxy:=this;
20         if length(myClients)<limit
21         then myClients:=append(myClients,cl)
22         else if nextProxy==null
23             then nextProxy:= new Proxy(limit,s) fi;
24             lastProxy:=nextProxy.add(cl) fi;
25         put lastProxy}
26     Void publish(Fut[News]fut){ var News ns;
27         ns :=get fut; // wait for the future
28         myClients!signal(ns); // multi-cast the result
29         if nextProxy==null
30             then s!produce() else nextProxy!publish(fut) fi}}
31 // Wrapper for the news producer.
32 class Producer(NewsProducerI np)implements ProducerI{
33     News detectNews(){ News news;
34         news:=np.getNews(); put news}}
35 // Local FIFO news producer queue.
36 class NewsProducer implements NewsProducerI{
37     List[News]requests:=Nil;
38     Void add(News ns)
39         {requests:=append(requests,ns)}
40     News getNews(){ var News firstNews;
41         await requests /= Nil;
42         firstNews := head(requests);
43         requests:=tail(requests);
44         put firstNews}}
45 // Consumer of news items.
46 class Client implements ClientI{
47     News news;
48     Void signal(News ns){news:=ns}}

```

Figure 3: A simple subscription example.

```

1
2 class Proxy(Int limit,Service s) implements ProxyI{
3   List[ClientI] myClients:=Nil;
4   ProxyI nextProxy;
5   ProxyI add(ClientI cl){
6     // unchanged from figure 4
7   }
8   Void publish(Fut[News]fut){
9     myClients!signal(fut); // send future, no waiting
10    if nextProxy==null
11      then s!produce()
12      else nextProxy!publish(fut) fi}}
13
14 class Client implements ClientI{
15   News news;
16   Void signal(Fut[News] fut) {
17     news:=get fut}} // blocking wait

```

Figure 4: A flooding variation of the subscription example.

Service.produce asynchronously calls Proxy.publish, line 13 of Fig. 3
 Proxy.publish asynchronously calls Client.signal, line 9 of Fig. 4
 Proxy.publish asynchronously calls Service.produce line 11 of Fig. 4

Each pass around this cycle, the asynchronous call to Proxy.publish is processed as part of the cycle (step 3). However, each pass around this cycle also spawns an asynchronous call to Producer.detectNews that is not processed as part of this cycle, nor is there any attempt to synchronize this cycle with the completion of those calls to Producer.detectNews. Depending upon the speed of execution of the code along the path of the cycle, such a cycle can create an unbounded number of suspended calls to Producer.detectNews.

We call such sequences flooding-cycles. In this paper we present an algorithm to statically identify programs that contain flooding-cycles. This approach is conservative in that if it reports that a program is free from flooding-cycles then it is indeed free of such cycles, however, it may report flooding-cycles that are in fact bounded by program logic, not amenable to static analysis. It may also report flooding-cycles that do not in practice produce an increasing number of unprocessed calls due to the execution speed of the flooding-cycle.

In the version of the program in Fig. 3, the cycle identified above will not cause a problem, provided the Client objects are able to process these signal calls at least as fast as they are being generated. Our algorithm will alert the programmer to this situation, and the programmer can determine if there is a real problem here, possibly with the aid of some additional program instrumentation. This is discussed further in section 4.2.

In the version of the program in Fig. 4, rather than wait in Proxy.publish for the news to actually be produced as in the first version (`ns:=get fut`), the method Proxy.publish instead simply passes the future out to another asynchronous call (`myClients!signal(fut)`) in line 9 of Fig. 4, eliminating any

progress coordination between the cycle producing the `Producer.detectNews` and `Client.signal` calls and the processing of those calls. The flooding cycle in this version of the program is more likely to be a problem because the completion of the `Producer.detectNews` and `Client.signal` calls are not dependent simply on execution speed of some code but are dependent upon the arrival rate of news items and in practice will always result in the number of unprocessed calls quickly growing to system limits.

4. Identifying Flooding Cycles

Definition 1 (Flooding Cycle). *For a given method m , an execution is flooding with respect to a method m if there is an execution cycle, call it C , containing a call statement $o!m(\dots)$ at a given program location, such that this statement may produce an unbounded number of uncompleted calls to method m , in which case we say that the call $o!m(\dots)$ is flooding with respect to C .*

Flooding of this kind may depend on the individual execution speed of the concurrent objects, and may be difficult to detect even when testing complete systems, due to their inherent non-deterministic nature. They may show up for instance at times when the system load is high, and they may depend on the scheduling of tasks inside an object. Flooding cycles that exist regardless of scheduling are more serious, indicating a defective program. We call such flooding *strong flooding*, otherwise *weak flooding*, as defined more precisely in section 5.1.

The top-level view of an algorithm to detect flooding-cycles is shown in Fig. 6. We first create a control flow graph (CFG), see Fig. 5, considering individual control flow graphs for each method where the nodes are method start nodes, method calls, *get* statements, *await* statements, or *put* statements (including implicit *put* statements at the end of Void methods). Blocking method calls will be represented as an asynchronous method call followed immediately by a *get* (as explained in Sec. 2). Boolean *await* statements are represented as nodes in the graph since they may affect method completion. Await and get statements may be assumed to never block when in a cycle since we consider unbounded iterations of the cycle. All other statements are ignored.

We connect the individual CFGs for each method with call edges, replicating the method's CFG for each non-recursive call so that we can associate a specific call with a specific start/get/put (step 2 of Fig. 5). Note that the static analysis will not take object identity into account. This means that we do not distinguish between multiple calls to the same object and calls to multiple instances of the same class. Thus the program graph will be finite. In step 3 (of Fig. 5) we identify any cycles in the graph. Note, cycles can include call edges and flow edges. In step 4 we assign unique labels to *call*, *start* and *put* nodes. In step 5 we add edges from *put* statements to *get* or *await* statements that block on the value for that future. Since each call node gives a copy of the corresponding method CFG, textually unique call nodes will correspond to unique put nodes. Thus there will be only one put edge leading to a get/await node waiting on the future of a textually unique call node. Passing futures as parameters will not violate this (see the examples in figures 4 and 11), unless the actual future is external to the cycle. In general the computation of put edges can be non-trivial for programs with non-trivial assignments to futures variables (say depending

1. Build the individual *control flow graphs* (CFGs) for each method including an initial *start node* and a final *put node*, and a node for each *call*, *get*, or *await*.
2. Add call edges from call nodes to the start node of a *copy* of the corresponding method CFG, unless the call is recursive, in which case create a call edge to the existing start node.
3. Identify any cycles in the graph.
4. Assign each call node a unique label, and assign this label to the start and put node of the corresponding method CFG.
5. Use flow analysis to compute the *put* edges from *puts* to the corresponding *get/await get* node.

Figure 5: Control Flow Graph.

1. Consider a cycle C in the graph G resulting from Fig. 5 and mark all nodes in G as *not reachable*.
2. Starting with the entry point to the cycle, do a depth-first traversal of G and apply definitions 4 and 7 to mark nodes as *weakly-reachable* (WR), *strongly-reachable* (SR), or neither.
3. If the previous two steps result in any changes to *strongly-reachable* or *weakly-reachable*, go to step 2.
4. Report flooding of call n if $n \in (calls - comps)$ in the given cycle where $calls = \{n \mid call_n \in WR\}$ and $comps = \{n \mid put_n \in SR \vee get_{\{n\}} \in SR\}$

Figure 6: Top Level Algorithm for Detecting Flooding relative to a given cycle.

on conditional branching); and the set of put edges leading to a put/await node might not be a singleton set. By static approximation one may estimate a label set of a get/await node such that the node may only wait on a future from the set. For the purpose of this paper we consider singleton or empty label sets, reflecting unique or unknown put-get flows, respectively.

The graph resulting from figure 5 is used in the detection algorithm given in Fig. 6 to identify any flooding cycles. If there is a cycle that creates futures (including implicit Void-valued futures for trivial calls) that are not read in the cycle or by nodes that must be executed as a result of the cycle, then there is a flooding-cycle with respect to the call that produced the unread future.

4.1. Computing the calls and comps sets

The algorithm of Fig. 6 considers one cycle C (at a time), ignoring any external call initiating the cycle, together with surrounding parts of the program graph. The algorithm is based on a notion of weakly reachable nodes (WR_C) representing nodes that *may* be reached from C following flow or call edges, and a notion of strongly reachable nodes (SR_C) representing nodes that *must*

be reached from C . The algorithm computes two sets; the set of calls that *could possibly have occurred* during the cycle or outside of the cycle when *reachable* from the cycle, and the set of calls that *must have completed* during the cycle or outside of the cycle when *reachable* from the cycle without suspensions or blocking. These sets are denoted $calls_C$ and $comps_C$, respectively. The nodes are labeled as follows:

- We label the call nodes in the graph by a number representing their textual occurrence in the (sub)program.
- Every method body begins with a *start* node and ends with a *put* node, and those two nodes will have the same label as the caller. Because of graph expansion, the caller will be unique except in the case of a recursive call. In that case, the label is the label of the non-recursive caller except when the cycle being considered includes the recursive call edge. In the latter case the label will be that of the recursive caller.
- We mark each *get* node with the set of labels corresponding to any *call* nodes that may have generated the future that reaches the *get*, given by the labels of all put edges leading into the get node.

We write $call_n$, put_n , and $start_n$ where n is the label of the corresponding call, and get_s where s is a label set.

Definition 2 (Program Graph). *A program-graph G is a graph comprised of start, call, get, await, and put nodes, and flow, call, and put edges, constructed according to Fig. 5.*

Definition 3 (Flow Path). *A flow-path is a path made up of only flow edges.*

Definition 4 (Weakly Reachable). *Node N is weakly-reachable (WR) with respect to cycle C in graph G if*

1. $N \in C$, or
2. $\exists P \in C$ such that there is a path from P to N comprised solely of call and flow edges.

Definition 5 (Fork-Free-Reachable). *A node N is fork-free-reachable from node P if there is only one flow edge from P , and that edge goes to N .*

Definition 6 (Join-Free-Reachable). *A node N is join-free-reachable from node P if there is only one flow edge to N , and that edge comes from P .*

Definition 7 (Strongly Reachable). *A node N is strongly-reachable (SR) with respect to cycle C in graph G if*

1. **(cycle node)** $N \in C$, i.e., N is in the cycle
2. **(ordinary node)** N is not a get or await node, and is fork-free-reachable from some $P \in SR$
3. **(get node)** N is a **get** node with label set $\{n\}$; the put node with label n is SR; N is fork-free-reachable from some $P \in SR$
4. **(start node)** N has label n ; **get** $_{\{n\}} \in SR$ (or **await get** $_{\{n\}} \in SR$) and there is a call edge $P \rightarrow N$ for $P \in SR$.

5. (**backward put**) N is a **put** node with label n ; there is a **get** node $\in SR$ with label n , and $call_n \in SR$
6. (**backward propagation**) some $P \in SR$ is join-free-reachable from N
7. (**multiple paths**) N is a **put node** with label n ; the start node with label n is SR and for every flow path from that start to N , all **get/await** nodes are SR .

This notion of strong reachability reflects nodes that must be executed when the cycle executes, or immediately afterwards. If we ignore puts and awaits, we know that fork-free steps must be taken immediately (since there is no other choice), see case 2. The start of a method call depends on scheduling, thus we cannot in general be sure that the call starts immediately. However, if we know that the call has ended ($put_n \in SR$), we also know that it has started, see case 4. Furthermore the path that lead to the **put** node must have been executed, except for possible branching. A blocking **get** must execute when the corresponding call has ended, see case 3. (If the label set of a get node has more than one label, each label in the set should be in *comps* as calculated from the current SR nodes.) Conversely, if a **get** is SR the corresponding **put** must be SR .

Definition 8 (Weakly Reachable Calls). *For cycle C in graph G , $calls_C$ is the set of labels of all weakly-reachable call-nodes in G .*

Definition 9 (Strongly Reachable Completions). *For cycle C , $comps_C$ is the set of labels n given by $comps_C = \{n \mid put_n \in SR_C \vee get_{\{n\}} \in SR_C\}$.*

These definitions complete the formalization of the detection algorithm of Fig. 6. For each cycle C in the control flow graph of a given program, our algorithm calculates the set $calls_C - comps_C$, and reports flooding of each call with label n in this set, as stated in Fig. 6.

4.2. Applying the Algorithm to the Example

Figures 7 and 8 show the call and comp sets for the two versions of the publish/subscribe problem above. To conserve space, all method names are abbreviated to the first letter of the class and the first letter of the method except that we use X for the class *Proxy* to further disambiguate it from *Producer*. For example, *Producer.detectNews* is Pd and *Proxy.publish* is written Xp .

Fig. 7 contains two cycles involving only flow edges and call edges. The call to *Client.signal* (Cs) is being flooded by both cycles. This does not produce an actual flood because the amount of work required by the Client to complete a signal call is trivial and thus the Client objects easily keep up with the calls. Also, the rate of execution for cycle A is limited by the actual arrival of news items from the *NewsProducer* (**await** requests $\neq Nil$), which further limits the rate at which this cycle generates asynchronous calls to *Client.signal*. Also, although not observed by our algorithm, cycle B is in fact finite, as it walks down the chain of Proxies. We call this *weak flooding*, i.e., flooding that is harmless, given underlying fairness of concurrent objects and fair scheduling of tasks within an object.

In the modified version of the program as reflected in figure 8 there is an additional flood of Pd (*Producer.detectNews*) by both cycles. This flooding-cycle is more serious, and in practice will flood the system almost immediately.

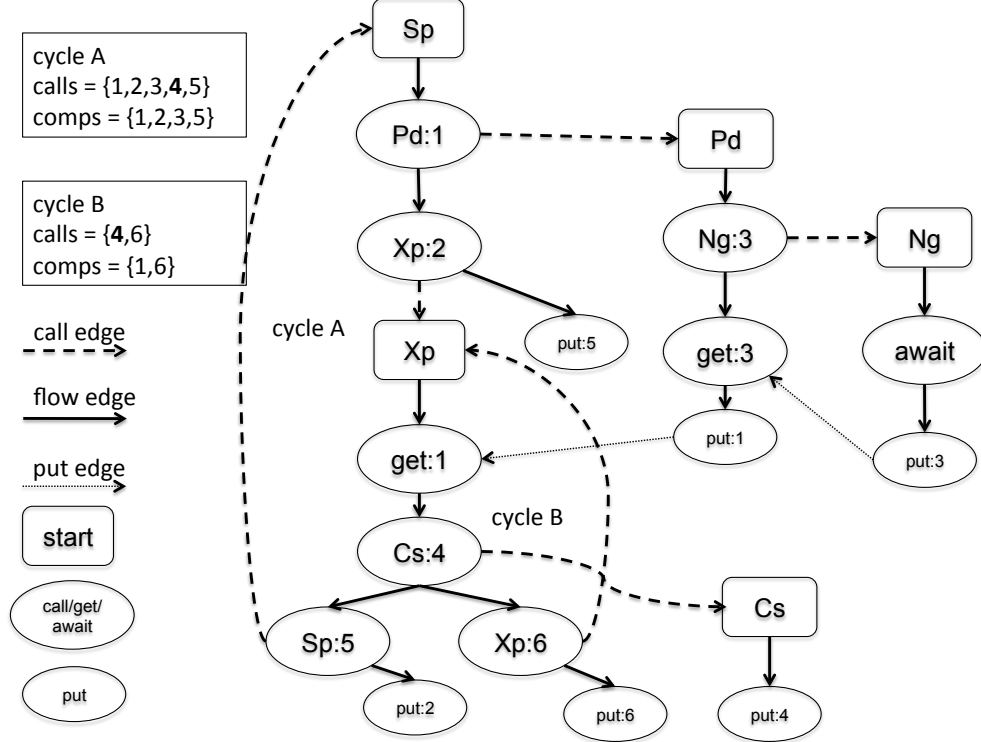


Figure 7: The graph and call/comp sets for the original version of the program (Fig. 3).

Unlike in version 1, there is no *get* regulating the speed at which cycle *A* cycles. Furthermore, the *Pd* calls are dependent upon the arrival of news items not simply limited by processor execution speed, as indicated by the presence of the Boolean *await* in *Ng* (News.getNews). This represents *strong flooding*.

The notions of strong and weak flooding are formally defined in the next section (5.1).

5. Main Results

In order to discuss the soundness of the presented algorithm with respect to the operational semantics, we need a notion of *executions*. For a given program (and starting object) the operational semantics from 7 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]*. In the following we will be concerned with infinite executions, since terminating executions are not flooding. 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*.

Definition 13 (Strongly Flooding Execution). *An execution has strong flooding with respect to a method m if it is flooding with respect to m and the execution has favorable scheduling of all objects.*

If there is an execution of a given program which is flooding with respect to a method m , our algorithm detects flooding of a call to m (no false negatives). We state this main result through two theorems, expressing detection of flooding cycles and more generally with respect to flooding executions.

Theorem 1 (Flooding with respect to a Cycle). *If $call_n$ is flooding with respect to cycle C for some execution of a given program, then flooding is detected by our algorithm.*

Theorem 2 (No false negatives). *If there is an execution of a given program which is flooding with respect to a method m , our algorithm detects flooding of a call to m , or of a call m' such that flooding of m' implies flooding of m .*

In section 6 we will discuss detection of strong flooding. Before proving the above theorem, we note that static analysis cannot in general determine whether the same or different objects are executing different processes, and we cannot in general distinguish futures from different cycle iterations.

For instance, an execution may be flooding even if each object has a bounded number of uncompleted processes, but this would require an unbounded number of objects. Such a case is illustrated in Fig. 9. To avoid such cases we assume that *an execution has a bounded number of objects*. (We may add that our simple notion of time as an index in the interleaved execution sequence is not suitable for discussing flooding in the presence of an unbounded number of objects.)

1	Void cycle() {x:=new C(); f:= x!m(); this!cycle(); put void}
2	Void m() {...; put void}

Figure 9: A minimalistic example. Here flooding is detected as the put node of m is not SR . However, no single object will be flooded.

Under this assumption an execution is flooding if and only if there is an object with an unbounded number of uncompleted processes. However, the assumption of bounded number of objects may lead to false positives as illustrated in figure 10. In each iteration of method *cycle* the get statement will wait for the next iteration to complete, so each iteration is blocked waiting for the next iteration. Each object executing an iteration is blocked and may not perform any later iteration. Thus an unbounded number of iterations requires an unbounded number of objects, say inserting `x:= new C();` before the call, which results in an unbounded number of executions of *cycle*, i.e., flooding. We may also conclude that if the number of objects is bounded, the same object must be used twice, resulting in a deadlock, but not flooding. A suspending get would give flooding even with a finite number of objects, even just one, inserting `x:=this;` before the call. The detection algorithm will not detect the put node as SR , and thus conclude possible flooding.

5.2. Lemmas and Proofs

Lemma 1. *If node $call_n$ is flooding with respect to cycle C in graph G then there must be a call chain from a call in C to $call_n$.*

1 Void cycle(){start₁; f:= x!cycle()₁; **get** f₁; **put** void₁}

Figure 10: A minimalistic example, including labels on call, start, put and get nodes.

Proof. This lemma follows directly from definition 1. ■

Lemma 2. *If node $call_n$ is flooding with respect to cycle C in graph G then $n \in calls_C$.*

Proof. This lemma follows directly from definitions 4 and 8. ■

Definition 14 (Efficiently Executable). *We say that a statement s is efficiently executable if it is executed within a time bound, i.e., for every execution there is a bound b such that whenever s is the next statement of an object o at time t , the execution of s is completed before time $t + b$.*

The bound may not depend on t , but may depend on bounded factors such as the particular object and statement.

Lemma 3 (Efficiently Executable Statements). *All basic statements in our language, including enabled suspension/get statements, are efficiently executable.*

The lemma does not say anything about non-enabled gets, await statements that are not initially enabled, as well as the first statement of method bodies.

Proof. This result follows by the operational semantics, considering each such basic statement. In our semantics each object is executing independently of other objects. The interleaving is unspecified, but we assume that each object is not unboundedly slower than the other objects (i.e. fairness among the executing objects), otherwise, trivial flooding would occur. One execution step of an object is considered taking one abstract time unit. An object o is executing one process at a time, unless idle, and each process defines a next statement. Consider now all cases of a next statement for o : Skip, assignment, call, new, and return can be executed in one unconditional step (since expressions are terminating). An if statement can be executed in one step by either rule if-true or if-false (defined in the appendix). A **get** or **await** statement can be executed in one step, if enabled. This proves our lemma since unbounded overtaking of an enabled o step by other objects is not allowed (due to inter-object fairness). ■

It follows that a (finite) sequence of efficiently executable statements will also be efficiently executable. We note that a blocking **get** may take an unbounded number of steps. This also means that incoming calls as well as enabled processes in the process queue may in general take an unbounded number of steps to get started, depending on the scheduling, if a blocking process comes first. Thus non-enabled get/await/await get statements are not in general efficiently executable.

Definition 15 (Efficiently Executable Relative to Cycle). *We say that a statement/method execution caused by a cycle iteration is efficiently executable relative to the cycle if it is executed within a bound after the end of the iteration, and the bound does not depend on that particular iteration.*

Flooding of a method means that each of an unbounded number of cycle iterations causes an execution of the method that is not efficiently executable relative to the cycle.

We next prove the following lemma:

Lemma 4. *If node N is strongly-reachable (SR) in graph G with respect to cycle C then the execution of N is efficiently executable relative to C .*

Proof. We will use structural induction on the set of SR nodes as defined by definition 7 to prove that Lemma 4 holds for all SR nodes.

Base case: Consider a node in C (definition 7.1). Clearly it must be executed as part of the iteration, and is therefore efficiently executable relative to C .

Induction step: Assume Lemma 4 holds for SR nodes up to a given structural SR complexity. We will show that applying each of the subparts of definition 7 defines a strongly reachable node N that is efficiently executable relative to C .

7.2: N is not a blocking or suspending node. P is SR and efficiently executable relative to C by the induction hypothesis. The only way to continue execution of the current object is to execute N , which is non-blocking and efficiently executable. We next use the fact that extending a finite sequence of efficiently executable statements by one more efficiently executable statement results in an efficiently executable statement sequence. Therefore N must be efficiently executable relative to C .

7.3: The argument is the same as for the previous case with the addition of the premise that put_n is SR, and therefore efficiently executable relative to C by the induction hypothesis. Since the only way to continue the execution of the method containing P is N and N is unblocked due to the completion of its future, N must be efficiently executable relative to C .

7.4: From the premise that get_n is SR, we have $n \in comps_C$. Then the method corresponding to $start_n$ has finished or will finish efficiently, and since it cannot finish without starting, the node $start_n$ must already have executed or will execute efficiently. By the induction hypothesis, the corresponding node $call_n$ is SR and must be efficiently executable relative to C , thus the node $start_n$ must also be efficiently executable relative to C .

7.5: The nodes $start_n$ and P (a get_n node) are SR and are efficiently executable relative to C (by the induction hypothesis). But get_n can only execute if put_n has executed, thus the put_n node N must have completed, and is therefore efficiently executable relative to C , provided the execution of get_n is from the same cycle iteration as that of put_n .

It could be that the get_n is waiting for a future of an earlier or later iteration, and that the get node is not efficiently executable (otherwise it is not a problem to consider it SR). Consider first a blocking **get** waiting for a future of an earlier iteration. The put_n is SR regardless of the execution of the get_n node, therefore in each cycle iteration the execution of put_n is SR, and thus efficiently executable relative to that iteration. The put_n is also efficiently executable relative to a later iteration, since the end of a cycle iteration is before the end of the next one. Thus the get_n is enabled (from the previous iteration's put_n) and efficiently executable.

Consider next a blocking **get** waiting for a future of a later iteration. There must be an unbounded number of such cases, otherwise we may ignore them. This will either make the program block or make it slower than if the future was the one from this iteration. Outside the flooding method m , a **get** on such

a future will therefore not be a problem. However, inside m it could be that the blocking **get** is causing the flooding. But the assumption of a bounded number of objects implies that the same object is being flooded by uncompleted m executions. But there cannot be an unbounded number of iterations where the object executing m is waiting for an m completion by the same object caused by a later iteration. This would give blocking of the cycle and therefore not be flooding.

7.6: The only way to get to node P is through N and by assumption P is SR and efficiently executable relative to C . Thus the only way for N to not be efficiently executable relative to C would be for P to be the first node in its method. But this is impossible.

7.7: Since each node in each path leading to N is SR and thereby efficiently executable, then regardless of the path taken at run-time, the final step to N must be efficiently executable relative to C . ■

5.3. Proof of Theorem 1: Flooding with respect to cycles is detected

Proof. If there is a flooding call, $call_n$, where $n \notin calls_C - comps_C$, then either

- a) $n \notin calls_C$, or
- b) $n \in calls_C$ and $n \in comps_C$.

By Lemma 1, there must be a call chain starting on the cycle that leads to $call_n$ and by Lemma 2 $n \in calls_C$. Therefore if the theorem does not hold, it must be because $n \in comps_C$ (and shouldn't be).

Let $n \in calls_C$ and $n \in comps_C$. We need to prove there is no flooding of $call_n$. By definition 9 put_n or $get_{\{n\}}$ are SR . In the former case the path to put_n is efficiently executable relative to C by lemma 4. In the latter case the path to $get_{\{n\}}$ is efficiently executable relative to C , and since put_n must have happened earlier, the path to put_n is also efficiently executable relative to C . There cannot be flooding of the method called by $call_n$, because flooding of n means that each of an unbounded number of cycle iterations causes an execution of the method which is not efficiently executable relative to the cycle. ■

5.4. Proof of Theorem 2: No false negatives are detected

Proof. Assume that there is a system execution E of a given program with flooding with respect to a method m . We need to prove that flooding of a call to m is detected (or of a method m' causing the flooding of m).

Consider a system execution E that is flooding with respect to method m . Since there are finitely many m calls in the code, the execution must also be flooding with respect to one of these calls, say labeled n . Thus it suffices to consider each textual call rather than each method. Let us assume that $call_n$ is flooding in execution E . This call must be caused by a cycle C , repeated without bounds. (The cycle may be interleaved with other flooding or non-flooding cycles.) Thus the call $call_n$ must occur in the iteration of the cycle, or in a method called (directly or indirectly) from the iteration, i.e., there must be a *call path from the cycle to the call*. We may assume that there is no flooding of an earlier call in the call cycle.

Thus flooding with respect to m in the sense of definition 11 reduces to flooding with respect to a cycle in the sense of definition 1. The theorem therefore follows by theorem 1. ■

Note that a cycle found by our analysis need not reflect a real cycle in an execution. For instance, if a cycle goes through a blocking **get** waiting for a future that in reality must be generated by the same object, this would in fact not be a cycle in the actual execution. Our analysis will pretend all method executions are done by different objects, thus the blocking situation is not discovered.

6. Extensions

The above detection algorithm for weak flooding may also be used to over-approximate strong flooding. However, for strong flooding, we will here discuss a tighter detection algorithm.

In the above detection of weak flooding, there is no need for an intra-object fairness condition, since we look at processes started during a cycle as well as efficient continuations of these. Non-enabled steps are not considered, nor calls leading outside the cycle unless they are known to complete. For detection of strong flooding we may assume favorable scheduling, including fairness of incoming calls as well as suspended calls. We first note that a blocked object (say waiting forever in a **get**) will have favorable scheduling with our definition since there cannot be an unbounded number of scheduling opportunities due to the blocking. Therefore permanent blocking of an object may lead to strong flooding of that object. However, this is a kind of deadlock problem that can be detected by other kinds of analysis, and is not primarily a flooding problem. We therefore assume below that *a flooding object is not dead* in this sense.

We now investigate tighter detection of suspending **gets** as well as of strong flooding.

6.1. Discussion on Suspending Get

An enabled suspending **get** might be chosen immediately, and thereby behave like a blocking **get**, or it may be chosen after a delay, letting other (enabled) processes go first. In the worst case it might never be chosen, for instance when the object constantly receives new method executions to execute. In our analysis the former (optimistic) case can be reflected by treating a suspending **get** as a blocking **get**, and the latter (pessimistic) case can be reflected by treating a suspending **get** as a Boolean await, since the analysis will never be sure about termination of a Boolean await, unless its termination follows from other reasons (backward propagation or being part of the considered cycle).

The above detection algorithm is doing the latter. While sound, this approach leads to over-approximation. In practice, a suspending **get** is often efficiently executed once enabled, thereby resembling a blocking **get**. We therefore reconsider the treatment of suspending **gets**, trying to provide a tighter detection algorithm under reasonable conditions. As illustrated by the example sketched in Figs. 11 and 12, we lose soundness by treating suspending **gets** as blocking **gets**. The sketch includes start nodes. Call labels are shown as indexes on call, start, and put nodes. Other details such as declarations of fields x and y are not shown. In the example, the suspending **get** could cause flooding; and even if enabled, it may be unboundedly delayed. In particular if x and y refer to the same object, the future f will be resolved before m starts since the *cycle* execution must terminate before the object is free to execute m , and at the

```

1 Void cycle() {start2; x!m(myfuture)1; put void2}
2
3 Void m(f) {start1; y!cycle()2; await get f; put void1}

```

Figure 11: An example sketch demonstrating the unsoundness of treating a suspending **get** as a blocking **get**.

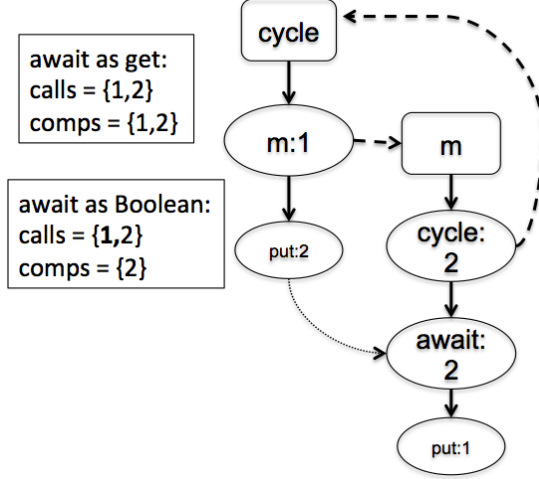


Figure 12: The graph and call/comp sets for the sketch from Fig. 11 treating suspending **get** (await) as a blocking **get** and as a Boolean await.

suspension point there is always another incoming call waiting. Our algorithm will report flooding when the suspending **get** is treated as a Boolean **await**, but not when treated as a blocking **get**, since in this case the analysis will make a put-get edge to the get node from *put void₂*, which is *SR*, thereby also making *put void₁* strongly reachable, which means that the set $calls_C - comps_C$ is empty and no flooding is detected.

In order to characterize the two approaches, we first observe that the set of weakly reachable nodes is not affected, thus $calls_C$ is not affected by the choice of approach. Next we observe that increasing the set of strongly reachable nodes results in more labels in $comps_C$, and thus fewer labels in $(calls_C - comps_C)$, so less flooding reported. In practice it is often the case that an enabled *suspending get* is efficiently executed, and we would like to investigate conditions that ensure soundness of treating them as blocking **gets**.

Consider an execution flooding with respect to a method m . There must be a cycle C with a call path to m . Consider a suspending **get** node that would be classified as *SR* when being a blocking **get** but not when being treated as a Boolean **await**. Such a statement cannot be on the cycle and it cannot be on the call path to the flooded method m . The suspending **get** is either part of the flooding method m or not. In either case $n \in comps_C$ where the **get** is indexed by n .

If the **get** is not part of the method m , it must then be true that an unbounded delay of the suspending **get** does not influence the flooding of m .

If the **get** is part of the method m , the suspending **get** statement could cause the flooding by never continuing after the suspension point, leaving an unbounded number of m processes waiting in the suspension point, assuming a bounded number of objects. If the object is not dead (blocked forever), this means that new calls on the object are given priority over old enabled suspension points, which is not likely to happen in a realistic implementation. If we assume that suspended and enabled processes *cannot be unboundedly delayed*, i.e.:

*an object cannot have an unbounded number of suspended processes
when these are continuously enabled,*

then there is no flooding in this case, and thus treating the suspending **get** as a blocking **get** is sound. This assumption holds for a large class of system implementations (for instance in the case of favorable scheduling with objects that are not dead or stuck in a process). For such implementations we may treat a suspending **get** as a blocking **get** in the analysis. We may conclude that with the assumption of a bounded number of objects and the above assumption of bounded delay, it is sound to analyze suspending calls as blocking calls.

6.2. Detection of Strong Flooding

We may now restrict ourselves to executions with favorable scheduling, and as above we may ignore possible flooding of dead/deadlocked objects.

In the case of strong flooding there must be a call m such that either an unbounded number of m calls are never started, or that an unbounded number of m calls are suspended and never continued. With a bounded number of objects, there must be at least one object which is flooding and not dead. By the assumption of favorable scheduling, new calls must be started and enabled continuations must continue without being unboundedly delayed. Thus we assume that start nodes are efficiently executable (once the call has taken place) and also that continuously enabled await statements are efficiently executable.

We may therefore modify the detection algorithm as follows: we generalize the definition of strongly reachable, replacing case 3 (**get node**) by

- 3 (**await/get node**) N is SR if it is a **get** node or an **await get** node with label set $s \subseteq \text{comps}_C$, and N is fork-free-reachable from some $P \in SR$

and case 4 (**start node**) of the definition of strongly reachable is replaced by

- 4 (**start node**) N is SR if there is a call edge $P \rightarrow N$ for $P \in SR$.

Thus a start node N is strongly reachable if there is a call edge $P \rightarrow N$ and P is strongly reachable, and an await get node is strongly reachable if all labels are in comps_C . The resulting notion of SR is denoted SR' and similarly comps' .

6.3. Detection of Strong Flooding in the Example

For the two versions of the publish/subscribe example consider again figures 7 and 8 which show detection of weak flooding. The first version of the example we have that the node Cs is SR' and therefore also node **put** : 4. For both cycles the set calculated by $\text{calls} - \text{comps}'$ is empty. Thus there is no case of strong flooding detected in this version.

For the modified version of the example we have that node **put** : 3 is not SR' due to the preceding await node. As a consequence neither node **put** : 1

nor `put : 4` are SR' , and for both cycles the set calculated by $calls - comps'$ is unchanged. Thus we detect strong flooding in cycle A with respect to both call 1 and call 4, i.e., *signal* calls and *detectNews* calls, and that there is strong flooding in cycle B with respect to call 4. This confirms the observation that the modified version has serious flooding. We also see that the detected cases of serious flooding are real.

7. Conclusion

In this paper we have presented an algorithm to statically detect over-eager concurrency in programs that use active objects and asynchronous method calls, such that one or more objects may be overwhelmed with more calls than it is able to handle. We call this over-eager concurrency flooding. We distinguish between two types of flooding, strong and weak. Weak flooding may depend on the underlying scheduling policy, whereas strong flooding persists even with favorable scheduling. We presented a proof that our algorithm is sound in the sense that if there is flooding, our algorithm will report it. The treatment includes a construct for active waiting for a future (the `get` statement).

The algorithm does not detect whether two method executions are performed by the same object or not, and it is not able to analyze whether a Boolean condition occurring in an if-test or `await` statement is satisfied. It may therefore detect possible flooding when there is no real flooding (over-approximation), for instance when there is an unbounded number of objects, and when the program uses passive waiting for a future (suspending `get` statement). We therefore assume a bounded number of objects at run-time. In order to increase the power of the algorithm we have identified an assumption under which it is sound to treat suspending `get` statements as blocking get statements, thereby significantly reducing the amount of over-approximation.

For implementations of active objects and asynchronous method calls with underlying fair scheduling, the concept of strong flooding is more interesting than weak flooding. We have shown how to modify the detection algorithm so that it detects strong flooding based on the assumption of favorable scheduling. This work assumes absence of deadlock.

We have considered a general communication model with a high degree of flexibility in communication and synchronization, based on local programming decisions. Concurrent objects that work well in one environment may flood or deadlock in another environment, since these problems are sensitive to whether calls from the environment use passive or active waiting, something which is not apparent in local reasoning. For instance, the strong flooding problem caused by the modified version of the *Proxy* class cannot be detected from local reasoning of that class alone. It is easy to unintentionally create programs that deadlock or flood, and therefore static detection of flooding (as well as deadlock) is valuable.

Future work includes improvements with respect to strong reachability of conditional `await` statements, combining static detection of progress properties and static detection of object disjointness and object aliasing. Detection of terminating cycles would further improve the analysis since termination eliminates flooding. In the example, cycle B terminates and detection of this would eliminate a (duplicate) report of weak flooding. An integration of detection of

strong flooding and deadlock detection would eliminate the need for the assumption of absence of deadlock. Finally we would like to investigate the algorithm on a larger set of case studies and to characterize its completeness.

References

- [1] G. Agha, S. Frolund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(2):3–14, may 1993.
- [2] G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1986.
- [3] Arvind and R. S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. on Computers*, 39(3):300–318, Mar. 1990.
- [4] D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer-Verlag, 2004.
- [5] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.
- [6] O.-J. Dahl. *Verifiable Programming*. International Series in Computer Science. Prentice Hall, New York, N.Y., 1992.
- [7] F. S. de Boer, M. Bravetti, I. Grabe, M. D. Lee, M. Steffen, and G. Zavattaro. A petri net based analysis of deadlocks for active objects and futures. In C. S. Pasareanu and G. Salaün, editors, *Formal Aspects of Component Software, 9th International Symposium, FACS 2012, Mountain View, CA, USA, September 12-14, 2012. Revised Selected Papers*, volume 7684 of *Lecture Notes in Computer Science*, pages 110–127. Springer, 2012.
- [8] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proceedings of the 16th European conference on Programming, ESOP’07*, pages 316–330, Berlin, Heidelberg, 2007. Springer-Verlag.
- [9] C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.
- [10] C. C. Din, J. Dovland, and O. Owe. Compositional reasoning about shared futures. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Proc. International Conference on Software Engineering and Formal Methods (SEFM’12)*, volume 7504 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2012.
- [11] C. C. Din and O. Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *Journal of Logical and Algebraic Methods in Programming*, 83(5-6):360–383, September 2014.

- [12] P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(1):6, 2012.
- [13] I. Grabe. *Static analysis of unbounded structures in object-oriented programs*. PhD thesis, Doctoral Thesis, Leiden University, 2012.
- [14] R. H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.
- [15] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15(3):253–285, Aug. 1997.
- [16] C. E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.
- [17] E. B. Johnsen. Separating cost and capacity for load balancing in ABS deployment models. In E. Giachino, R. Hähnle, F. S. de Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects - 11th International Symposium, FMCO 2012, Bertinoro, Italy, September 24-28, 2012, Revised Lectures*, volume 7866 of *Lecture Notes in Computer Science*, pages 145–167. Springer, 2012.
- [18] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.
- [19] E. B. Johnsen and O. Owe. A dynamic binding strategy for multiple inheritance and asynchronously communicating objects. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. 3rd International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 274–295. Springer, 2005.
- [20] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
- [21] E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous method calls. In *Proc. 5th International Workshop on Rewriting Logic and its Applications (WRLA’04)*, Mar. 2004.
- [22] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. T. Tarifa. Dynamic resource reallocation between deployment components. In J. S. Dong and H. Zhu, editors, *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, volume 6447 of *Lecture Notes in Computer Science*, pages 646–661. Springer, 2010.

- [23] E. B. Johnsen, O. Owe, and A. B. Torjusen. Validating behavioral component interfaces in rewriting logic. *Fundam. Inf.*, 82(4):341–359, Feb. 2008.
- [24] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.
- [25] B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI’88)*, pages 260–267. ACM Press, June 1988.
- [26] C. McDowell and O. Owe. Towards a light-weight approach for concurrent active objects in java. Technical report, UCSC, Santa Cruz, USA, 2015.
- [27] B. Nobakht, F. S. de Boer, M. M. Jaghoori, and R. Schlatte. Programming and deployment of active objects with application-level scheduling. In *SAC’12*, pages 1883–1888. ACM, Mar. 2012.
- [28] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala. A comprehensive step-by-step guide*. Artima Developer, 2008.
- [29] G. D. Plotkin. A structural approach to operational semantics. Research Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, Sept. 1981.
- [30] J. Xu, Y. Lei, R. H. Carver, and D. C. Kung. A lightweight, static approach to detecting unbounded thread-instantiation loops. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10. IEEE Computer Society, 2015.
- [31] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA ’86). Sigplan Notices*, 21(11):258–268, Nov. 1986.

Appendix A: Operational Semantics

We define the operational semantics of the core language by a transition system, presented in the style of structured operational semantics (SOS) [29]. A global system state (*configuration*) consists of objects, messages, and futures, each identified by unique identifiers. A system state g is therefore captured as a multiset of objects, messages, and futures, each identified by its unique identifier, letting $id : \mathbf{ob}(\text{state}, \text{code})$, $id : \mathbf{call}(\text{callee}, \text{method}, \text{args})$, and $id : \mathbf{ret}(\text{value})$, represent an *object*, an *invocation message*, and a *future*, respectively. Here code denotes a sequence of statements followed by a **put** statement, representing the remaining code to be executed by the currently active method call, or is **idle**. In the operational semantics each new generated object is given a new object identity, using the special variable `nextObj` of the parent object. Similarly each new method call generates a new future identity, using the special variable `nextFut` of the caller object. A given future identifier is associated with an invocation message after the call is made and before its execution starts, and is associated with a future after the method execution has terminated with a resulting return value. The special constant `null` may not identify an object. Data types and associated functions have a fixed interpretation.

Map Notation. A finite mapping is seen as a set of bindings $z_i \mapsto \text{value}_i$ for a finite set of disjoint identifiers z_i , the *domain*. The empty map is denoted ε . Map look-up is written $M[z]$ where M is a mapping and z an identifier. The notation is lifted to expressions, letting $M[e]$ mean the expression e evaluated in the state given by M . Map composition is written $M + M'$ where bindings in M' override those in M for the same identifier. A map update, written $M[z \mapsto d]$, is the map M updated by binding z to d , i.e., $M[z \mapsto d]$ is the same as $M + [z \mapsto d]$. For an expression e , the notation $M[z := e]$ abbreviates $M[z \mapsto M[e]]$.

The state of an object is given by a twin mapping, written $(a|l)$, where both a and l are mappings: the “attribute state” a is the state of the field variables (including `this`, `nextFut`, `nextObj`, the class parameters $\overline{cp}$, and the local process queue PQ), and the “local state” l is the state of the parameters and local variables (including the implicit parameter `myfuture`). An idle object will have an empty local state (ε), and thus has the form $\mathbf{ob}((a|\varepsilon), \mathbf{idle})$. Look-up $(a|l)[z]$ is simply given by $(a + l)[z]$. The notation $(a|l)[v := e]$ abbreviates **if** $v \text{ in } l$ **then** $(a|l[v \mapsto (a|l)[e]])$ **else** $(a[v \mapsto (a|l)[e]]|l)$, where in is used for testing domain membership. The process queue PQ is the queue of suspended processes, of form $(l, \overline{s})$. The operations $\text{enq}(PQ, p)$ and $\text{deq}(PQ)$ 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 is *enabled* if it starts with an enabled statement. A conditional **await** statement is enabled if the condition evaluates to true, and a **get** statement, as well as an **await get** statement, is enabled if the corresponding future value is in the configuration. All other statements are enabled.

Definition 16 (Configurations). *A configuration is a multiset of messages of the form $u : \mathbf{call}(o, m, \overline{d})$, futures of the form $u : \mathbf{ret}(d)$, where $u \in \text{Fid}$, and objects of the form $o : \mathbf{ob}(\delta, \overline{s})$ where $o \in \text{Oid}$, $o \neq \text{null}$, and $(\text{this} \mapsto o) \in a$. The state δ of an object has the form of a twin mapping $(a|l)$.*

The class table may also be included in the configuration, associating the class name C with the relevant class information such as the attributes of the class and the set of methods, say $C : \mathbf{class}(att, ms)$, where the method definitions in a class are of the form, $(m, \bar{p}, l, \bar{s})$ where m is the method name, $\bar{p}$ is the list of formal parameters, l contains the local variables (including default values), and $\bar{s}$ is the code. However, as class tables represent static information, not changed at run-time, they are not explicitly included in the operational semantic rules.

Generation of fresh object and future identifiers is modeled by the initial algebra given by the constructors:

- $initO$ taking an object identity (the generating object) and returning an (initial) object identity,
- $initF$ taking an object identity (the generating object) and returning an (initial) future identity,
- $next_{Fid}$ taking a future identity (say the last generated one) and returning a future identity,
- $next_{Oid}$ taking an object identity (say the last generated one) and returning an object identity.

When no ambiguity arises, we omit the index on the $next$ function. A generator term, say $next(next(next(initF(o))))$, represents a unique future identity. Equality over generator terms is given by syntactic equality, thus local uniqueness implies global uniqueness since the generating object is encoded in the terms. The operational semantics uses an attribute $nextFut$, initialized to $initF(\mathbf{this})$, such that a fresh future identity is generated by $next(nextFut)$. Similarly, the attribute $nextObj$ is initialized to $initO(\mathbf{this})$, such that a fresh object identity is generated by $next(nextObj)$.

Operational Rules

The operational rules are given in Fig. 13 using $\rightarrow$ as the transition symbol. Conditional rules have an **if**-condition. Each rule deals with only one object o , and possibly messages and futures, reflecting that we deal with distributed concurrent systems communicating asynchronously. For disjoint *substates* g_1 and g_2 (i.e., disjoint multisets), $g_1 \ g_2$ denotes their parallel composition, using blank-space as a binary configuration constructor, which is associative, commutative and has the empty configuration ε as identity. In the operational rules we assume pattern matching modulo associativity, commutativity, identity (ACI). We use interleaving semantics, which may be defined by the general rule

$$\frac{g_1 \rightarrow g'_1}{g_1 \ g_2 \rightarrow g'_1 \ g_2}$$

Thus objects are concurrent in the sense that their executions are interleaved, and in each object, statements are executed sequentially.

Rules for skip, assignment and if-statements are straight forward. For instance, the rule $o : \mathbf{ob}(\delta, \mathbf{skip}; \bar{s}) \rightarrow o : \mathbf{ob}(\delta, \bar{s})$ expresses that the **skip** statement is consumed in one execution step. Variables are denoted by single

skip :	$o : \mathbf{ob}(\delta, \mathbf{skip}; \bar{s}) \rightarrow o : \mathbf{ob}(\delta, \bar{s})$
assign :	$o : \mathbf{ob}(\delta, v := e; \bar{s}) \rightarrow o : \mathbf{ob}(\delta[v := e], \bar{s})$
if-true :	$o : \mathbf{ob}(\delta, \mathbf{if } c \mathbf{ then } \bar{s1} \mathbf{ else } \bar{s2} \mathbf{ fi}; \bar{s})$ $\rightarrow o : \mathbf{ob}(\delta, \bar{s1}; \bar{s})$ $\mathbf{if } \delta[c] = \mathbf{true}$
if-false :	$o : \mathbf{ob}(\delta, \mathbf{if } c \mathbf{ then } \bar{s1} \mathbf{ else } \bar{s2} \mathbf{ fi}; \bar{s})$ $\rightarrow o : \mathbf{ob}(\delta, \bar{s2}; \bar{s})$ $\mathbf{if } \delta[c] = \mathbf{false}$
call :	$o : \mathbf{ob}(\delta, u := v!m(\bar{e}); \bar{s})$ $\rightarrow o : \mathbf{ob}(\delta[u := \mathbf{nextFut}, \mathbf{nextFut} := \mathbf{next}(\mathbf{nextFut})], \bar{s})$ $\delta[\mathbf{nextFut}] : \mathbf{call}(\delta[v], m, \delta[\bar{e}])$
start :	$u : \mathbf{call}(o, m, \bar{d})$ $o : \mathbf{ob}((a \varepsilon), \mathbf{idle})$ $\rightarrow o : \mathbf{ob}((a (l[\bar{p} \mapsto \bar{d}, \mathbf{myfuture} \mapsto u])), \bar{s})$ $\mathbf{where } m \text{ is statically bound to } (m, \bar{p}, l, \bar{s})$
continue :	$o : \mathbf{ob}((a \varepsilon), \mathbf{idle})$ $\rightarrow o : \mathbf{ob}((a[PQ \mapsto \mathbf{rest}] l), \bar{s})$ $\mathbf{if } \mathbf{deq}(a[PQ]) = ((l, \bar{s}); \mathbf{rest})$
return :	$o : \mathbf{ob}((a l), \mathbf{put } e)$ $\rightarrow o : \mathbf{ob}((a \varepsilon), \mathbf{idle})$ $l[\mathbf{myfuture}] : \mathbf{ret}((a l)[e])$
query :	$u : \mathbf{ret}(d)$ $o : \mathbf{ob}(\delta, [\mathbf{await } v := \mathbf{get } e; \bar{s})$ $\rightarrow u : \mathbf{ret}(d)$ $o : \mathbf{ob}(\delta[v \mapsto d], \bar{s})$ $\mathbf{if } \delta[e] = u$
new :	$o : \mathbf{ob}(\delta, v := \mathbf{new } C(\bar{e}); \bar{s})$ $\rightarrow o : \mathbf{ob}(\delta[v := \mathbf{nextObj}, \mathbf{nextObj} := \mathbf{next}(\mathbf{nextObj})], \bar{s})$ $\delta[\mathbf{nextObj}] : \mathbf{ob}(\delta_{\mathbf{init}}, \mathbf{init}_C)$
await :	$o : \mathbf{ob}(\delta, \mathbf{await } c; \bar{s})$ $\rightarrow o : \mathbf{ob}(\delta, \bar{s})$ $\mathbf{if } \delta[c] = \mathbf{true}$
suspend :	$o : \mathbf{ob}((a l), \bar{s})$ $\rightarrow o : \mathbf{ob}((a[PQ \mapsto \mathbf{enq}(a[PQ], (l, \bar{s}))], \varepsilon), \mathbf{idle})$ $\mathbf{if } \bar{s} \text{ starts with } \mathbf{await}/\mathbf{await get} \text{ and no other rule applies to } o$

Figure 13: Operational rules reflecting small-step semantics.

characters (the uniform naming convention is left implicit), and $\bar{s}$ denotes a statement sequence ending with a put.

Asynchronous method invocation is captured by the rule **call**. The generated future identity is locally unique, and also globally unique since the identity is given by a generator term embedding the parent object. The future identity generated by this rule is first bound to an invocation message, which is to be consumed by rule **start**. And a future value is generated upon method completion reusing the same future identifier. We assume that method names are unique in each class. In Rule **start**, we assume that m is bound to a method with local state l (including default values), parameters $\bar{p}$, and code $\bar{s}$. Note that parameters and the implicit parameter **myfuture**, which are read-only, are added to the local state in Rule **start**.

When there is no active code in an object, 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 future identity of the call is assigned to the implicit parameter **myfuture**. Method execution is completed by rule **return**, and a (resolved) future value is fetched by rule **query**. A query statement blocks until the corresponding future value is generated by rule **return**. Note that rule **query** does not remove the future unit from the configuration, which allows several objects to fetch the value of the same future.

Object creation is captured by the rule **new**. The generated object identity is locally unique, and also globally unique since the identity is given by a generator term embedding the parent object. The object identity generated by this rule is bound to the generated object. In Rule **new**, δ_{init} denotes the initial state (including the bindings $\text{this} \mapsto \delta[\text{nextObj}]$, $\bar{c}\bar{p} \mapsto \delta[\bar{e}]$), and default/initial values for the fields; and $init_C$ denotes the initialization statements of class C . We obtain an active object by letting $init_C$ initiate internal activity, using suspension or asynchronous local calls to allow the object to interleave continued internal activity with response to external calls. The initialization statements of the starting object of a program will typically create the other initial objects.

A suspending get statement **await** $v := \text{get } u$ is enabled if the query $v := \text{get } u$ is enabled (i.e., the future $u : \text{ret}(d)$ is in the configuration for some d). And **await** $v := \text{get } u$ is equivalent to $v := \text{get } u$ if enabled, as indicated by the optional **[await]** in rule **query**. In the case that an await/await get statement is not enabled, the current process is placed on the process queue (without removing the await statement) 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 intra-object scheduling.

The given language fragment may be extended with constructs for local (stack-based) method calls, e.g., by using the approach of [20]. As we focus on inter-object communication, this is omitted here. We omit the rule for the call statement $v := o.m(\bar{e})$ since this call is equivalent to an asynchronous call ($u := o!m(\bar{e})$) followed by a get statement, as explained in Sec. 2.

It is straight forward to see that the operational rules maintain the conditions of definition 16 and that each object o in a configuration may be reduced further, unless it starts with a non-enabled **get** or is idle without any enabled process in the queue PQ nor any incoming calls to o .