Deadlock detection of active objects with synchronous and asynchronous method calls [*]

Olaf Owe and Ingrid Chieh Yu
Department of Informatics, University of Oslo
E-mails: {olaf,ingridcy}@ifi.uio.no

Abstract

Open distributed systems are essential in today's software solutions. However, not all programming paradigms provide natural support for such systems. The setting of concurrent objects is attractive since it supports independent units of computation. In particular we consider concurrent objects communicating by *asynchronous method calls* supporting non-blocking as well as blocking method calls. In this setting waiting time can be reduced, allowing efficient cooperation between objects. With this concurrency model, deadlock is avoided if blocking calls are avoided. However, blocking calls are sometimes needed to control the order of computation. The non-hierarchical nature of concurrent objects systems gives rise to non-trivial deadlock situations. Deadlocks may occur if there is a call chain with at least one blocking call.We propose a method for static detection of deadlocks, and demonstrate its use on a non-trivial example.

1 Introduction

Distributed systems play an essential role in the modern society, and we depend on such systems in our daily lives concerning finance, medicine, or Internet services. Software errors in such systems could cause serious problems for their users. Distributed and interacting software are error prone, and non-trivial errors appear. One example of such errors is

[*]This work was done in the context of the EU projects FP7-610582 *Envisage: Engineering Virtualized Services* and FP7-ICT-2013-X *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations.*

This paper was presented at the NIK-2014 conference; see http://www.nik.no/.

deadlock, which describes a situation where several distributed units are waiting for each other, without any of them being able to proceed [1].

Deadlock analysis is difficult when the software is programmed in languages with low level synchronization primitives, such as locks or wait/notify mechanisms, where the control is potentially distributed over several units. It is therefore interesting to explore more high-level language mechanisms that may simplify deadlock analysis and at the same time allow efficient code. The setting of concurrent object systems is attractive since it supports independent units of computation. The actor model is adopted by several languages including Erlang [2] and Scala [3]. In particular we consider concurrent objects communicating by *asynchronous method calls* supporting non-blocking as well as blocking method calls. In this setting synchronization is high-level as explicit use of locks or signaling is not needed, and yet cooperation is efficient since blocking calls can be avoided. In the concurrency model considered, an object has its own virtual processor and processes its own methods. In order to allow process suspension, each object has a process queue consisting of remaining parts of method instantiations.

The non-hierarchical nature of concurrent objects systems gives rise to non-trivial deadlock situations. Deadlock is avoided if blocking calls are avoided. However, blocking calls are sometimes needed to control the order of computation, and deadlocks may then occur if there is a call chain with blocking and non-blocking calls, with at least one blocking call. For instance consider two classes, $C1$ with methods $m1$ and $m3$, and $C2$ with method $m2$, such that $m1$ calls $m2$, $m2$ calls $m3$, and $m3$ makes no further calls. Assume that method $m1$ is executed by an object $o1$, calling $m2$ on $o2$. If the call to $m2$ is blocking there is a real deadlock when $m2$ actually calls back to $o1$; and if the call in $m2$ is non-blocking, $o2$ is not blocked but the $m2$ call cannot complete. In contrast, if the call in $m1$ is non-blocking, there is no deadlock even if the call in $m2$ is blocking, since $o1$ is free to execute incoming calls (such as $m3$) while $o2$ is waiting. Some approaches (like [4]) are not able to detect deadlock freedom in the last case.

The contribution of this paper is to propose a way to statically detect potential deadlocks in this setting. We use a variation of ownership types to improve the deadlock analysis [5]. The approach is compositional in the sense that each class is analyzed independently of its environment apart from information associated with interfaces. The analysis is extensible in the sense that when new classes and subclasses are added, the results of the previous deadlock analysis is easily updated with new requirements reflecting possible new call chains. The analysis method can easily be done by hand (for limited program size) and can easily be automated. We demonstrate the strength of our analysis method on a non-trivial example.

$$\begin{array}{lcl@{\qquad}lcl}
T & ::= & I \mid \mathsf{Bool} & K & ::= & \mathbf{interface}\ I\ \mathbf{extends}\ \overline{I}\ \{\ \overline{D}\ \} \\
 & & \mid \mathsf{Int} \mid \mathsf{Void} & L & ::= & \mathbf{class}\ C\ \mathbf{implements}\ I\ \{\overline{T}\ \overline{f};\{t\}\ \overline{M}\ \} \\
v & ::= & f \mid x & M & ::= & D\ \{\overline{T}\ \overline{x}; t; \mathbf{return}\ e\} \\
e & ::= & v \mid \mathsf{self} & D & ::= & T\ m\ (\overline{T}\ \overline{x}) \\
 & & \mid \mathsf{null} \mid \mathsf{void} & t & ::= & t;t \mid v := \mathit{rhs} \mid e!m(\overline{e}) \mid \\
 & & & & & \mathbf{if}\ b\ \mathbf{then}\ t\ \mathbf{else}\ t\ \mathbf{fi} \mid \mathbf{skip} \\
 & & & \mathit{rhs} & ::= & \mathbf{new}\ C(\,) \mid e.m(\overline{e}) \mid \mathbf{await}\ e.m(\overline{e}) \mid e
\end{array}$$

Figure 1: The language syntax. C is a class name, I an interface name, and m a method name. Variables v are fields (f) or local variables (x), and e denotes side-effect free expressions over the variables and b expressions of Boolean type. Vector notation denotes collections, as in the expression list $\overline{e}$ (we write $\overline{T}\ \overline{x}$ to denote a list of variable declarations), and in declarations lists such as $\overline{K}, \overline{L}, \overline{D}$, and $\overline{M}$.

2 Language

We consider an object-oriented kernel language akin to Featherweight Java [6] and high-level Creol [7, 8] (without futures nor labels). The syntax for classes and interfaces in the language is given in Fig. 1. As in Creol, objects are executing concurrently and can be used to capture distributed units, while local data structures are defined by data types. As data types are not essential in the discussion of deadlock, we omit their syntax in Fig. 1. Interaction between objects is by method calls only. Neither remote access to fields nor shared variables is allowed.

A program consists of a set of interfaces and class definitions, followed by a method body. The class' constructor $\{t\}$ is executed upon object creation. We assume that programs are well-typed. A method body consists of a sequence of standard statements followed by **return** e, where e is the value returned by the method. The return type Void reflects a trivial return, where Void is a predefined type with one element, void.

We let pointers (references) to objects be typed by interfaces. If a class C implements an interface I then its instances may be typed by I; we say that instances of C *support* I. The language does not support remote access of variables since this would break with the interface encapsulation.

The language provides substitutability at the level of interfaces: *an object supporting an interface I may be replaced by another object supporting I* in a context depending on I. This substitutability is reflected in the semantics by the fact that late binding applies to all method calls, as the runtime class of an object reference is not in general statically known.

Each method invoked on the object leads to a new process, and at most one process is executing on an object at a time; the other processes in the object are suspended. We distinguish between blocking a process and releasing a process. Blocking is used for synchronization and stops the execution of the process, but does not let a suspended process resume. Releasing a process suspends the execution of that process and lets a suspended process resume. Thus, if a process is blocked there is no execution in the object, and if it is released another process in the object may execute. A *blocking method call* v:=o.m($\overline{e}$) immediately blocks the processor while waiting for a reply. A *non-blocking call* v:= **await** o.m($\overline{e}$) releases the processor while waiting for a reply, allowing other processes to execute. When the reply arrives, the suspended process becomes enabled and the evaluation may resume. When a reply is not needed, the *trivial call* o!m($\overline{e}$) causes the method to be called, but the current object (the caller) will neither block nor suspend. Thus a trivial call can always be executed. This approach to method-based interaction provides flexibility in the distributed setting: suspended processes or new method activations may be evaluated by the callee while the caller may either continue execution or decide to wait for a reply. Local calls are possible by means of remote calls to self.

The Publisher-Subscriber example

We consider a version of the Publisher-Subscriber example given in Fig. 2. This example has been used to illustrate the future mechanism [9], and is here adapted to the current communication primitives, without futures. In the example, clients may subscribe to a service and the service object then generates news and distributes news updates to clients subscribing to this service. When publishing news, the service objects delegate publishing to proxy objects where each object distributes the news to a bounded number of clients. As in [9] the Service object handles a subscribe request efficiently by delegating the time consuming parts to Proxy objects, and the proxies publish news to clients without use of blocking calls. This makes the cooperation efficient.

For convenience we omit the trivial return statements of Void method (i.e., **return** void). We lift the notation for trivial calls to lists of objects, letting l!m(..) be understood as a multicast when l is a list of objects. Internal data structures, such as lists, are defined by data types. We let the append function append an element to a list, and Nil denote the empty list.

The example illustrates all call mechanisms of the language, blocking, non-blocking, trivial, and self calls. The example does not cause deadlock; however, it is nontrivial to see that the add call in line 33 does not cause deadlock. And when trivial calls are turned into blocking calls, interesting deadlock situations appear. The analysis of this is discussed later.

```
interface ServiceI{
    Void subscribe(ClientI cl);
    Void produce()
}
interface ProxyI{
    ProxyI add(ClientI cl);
    Void start_publish()}
    Void publish(News ns) –– internally called
}
interface ClientI{
    Void signal(News ns)
}
interface ProducerI{
    News detectNews()
}
class Service(Int limit) implements ServiceI{
    ProducerI prod; ProxyI proxy; ProxyI lastProxy;
    {prod := new Producer(); proxy := new Proxy(limit,self,prod);
     lastProxy := proxy; self!produce()}

    Void subscribe(ClientI cl){lastProxy := lastProxy.add(cl)}

    Void produce(){proxy!start_publish()}
}

class Proxy(Int limit, ServiceI s, ProducerI prod) implements ProxyI{
    List<ClientI> myClients; ProxyI nextProxy; {myClients := Nil}

    ProxyI add(ClientI cl){
       ProxyI lastProxy := self;
       if length(myClients) < limit then myClients := append(myClients, cl)
       else if nextProxy = null then nextProxy := new Proxy(limit,s,prod) fi;
       lastProxy := nextProxy.add(cl) fi; put lastProxy}

    Void start_publish(){
       News ns = None;
       ns := await prod.detectNews();
       self!publish(ns);
       fi}

    Void publish(News ns){
       myClients!signal(ns);
       if nextProxy = null then s!produce() else
       nextProxy!publish(ns) fi}
}
```

Figure 2: Implementation of the Publisher-Subscriber example.

3 Deadlock

Deadlock is a situation where a number of objects are waiting for each other to finish some execution. This can be illustrated by a directed graph whose vertices are processes and whose edges represent the call wait-for relation. If this graph contains a cycle, the system is deadlocked. However, a call cycle with only non-blocking calls will not lead to deadlock. In a deadlock situation there must be a call cycle with at least one blocking call. Trivial calls do not cause waiting nor deadlock and need not be considered. We may choose to consider the chain starting in a blocking call.

Runtime Deadlock Cycle: A deadlock cycle in a runtime configuration is a cycle of N objects o_i $(0 < i \leq N)$ where $N > 1$ such that $o_1 = o_N$ and $o_1 \neq o_i$ for $1 < i < N$, and where o_i makes a call to o_{i+1} and where o_1 makes a blocking call.

In a deadlock cycle the objects waiting in blocking calls (at least one) cannot do anything and are deadlocked, while the objects doing non-blocking calls can process other enabled processes, but will never be able to continue with the suspended non-blocking call.

We are interested in detecting possible deadlock cycles by means of static analysis. We want to work with one class at a time, relying on interface information (and possible calls from interfaces). We assume that analysis is applied on well-typed programs. Thus call cycles will be approximated by call chains using a notion of *deadlock chain*, defined below. The purpose of this notion is that *if we can conclude by static analysis that a program has no deadlock chain, it will follow that the program has no real deadlocks, i.e., no runtime deadlock cycle* (soundness).

Initial observations

We focus on deadlock analysis and may ignore problems such as calls to null objects, which give run-time errors but not deadlock, and recursion, which may give non-termination but not deadlock. And we ignore communication problems due to faults in the underlying network connecting the objects. We assume that analysis is applied on well-typed programs and thus method-not-understood errors will not appear [7]. As we will see below, deadlock analysis may exploit knowledge about the callee in the special case where it is self, a fresh object, or an object younger than the caller.

Static detection of self, fresh and younger objects: For the purpose of this paper, we apply a simple static analysis of the notion of *fresh* and *younger* objects: A callee o is recognized as a *fresh C* object if each path leading to the call has a final update to o of the form o:= new C(..).

A callee o is recognized as *younger* than the current object (the caller) if it is fresh or o is not a parameter and all updates to o in the class are of the form o:= new C(..). The type of the callee is recognized as C when the same C appears in all o:= new C(..) statements in the class.

Similar static analysis may be done to detect that a caller is self.

Notation

We use the notation C : m $\rightarrow$ I : m' to denote that execution of method m of an object of class C may cause a call to m' on an object of interface/class I. Here m may be init to denote the initialization block of class C, and I may be self to denote a self call. Thus C : m $\rightarrow$ self : m' represents a special case of C : m $\rightarrow$ C : m'.

The calls C : m $\rightarrow$ I : m' and C' : m' $\rightarrow$ J : m" can be *chained* if either C' is I, C' is a class implementing (a subinterface of) I, or I is self and C' is C.

A *call chain* is a list of at least two calls where each call (apart from the last) can be chained with the next call, and a *call chain cycle* is a call chain where the last call can be chained with the first call.

Static Deadlock Chain: A *deadlock chain* is a call chain of length two or more (not counting self calls) starting with a blocking call in some class C, say C : m $\rightarrow$ I_1 : m_1, and ending with a call .. : .. $\rightarrow$ I_n : m_n where C is a class implementing I_n and where I_n is not self.

We may represent deadlock chains by a finite set of finite call chains. For any call chain with an inner chain cycle, the chain without the cycle will also be a call chain. Thus in a *minimal call chain* the same inner call may only appear once. Thus there can only be finitely many such minimal chains for a finite set of classes and methods.

- In a deadlock chain we can remove inner cycles (excluding the first and last call) of length two or more. The resulting chain is still a deadlock chain.[1]
- We may remove all calls (apart from the first) of form C : m $\rightarrow$ self : m, since the chain without the last call must then be a deadlock chain.
- The last call in a deadlock chain should have a callee that potentially could be the first object of the chain. Thus we may demand that the last call is not to a fresh object nor self.

[1]In order to have a chain, the callee interface in the call before the cycle may need to be lifted so that it is supported by the class of the call following the cycle.

- Calls to younger objects than the caller cannot in itself create a deadlock cycle. Thus we may demand that not all calls in the chain are to younger objects. In particular direct recursive calls to younger objects of the same class may be ignored.

For a given program we let $\mathcal{C}$ be the set of all call-tuples for the program, and $\mathcal{C}_b$ be the subset of blocking calls. The static calculation of minimal deadlock chains follows these steps (assuming that static knowledge of the subinterface and implements relations is available):

1. detect the set $\mathcal{C}$ of all calls $\mathsf{C} : \mathsf{m} \rightarrow \mathsf{I} : \mathsf{m}'$ in the code for each class C and method m, with I' as small as possible (self if possible).
2. remove all calls $\mathsf{C} : \mathsf{m} \rightarrow \mathsf{self} : \mathsf{m}$ from $\mathcal{C}$.
3. remove all calls $\mathsf{C} : \mathsf{m} \rightarrow \mathsf{C} : \mathsf{m}$ from $\mathcal{C}$ when the callee is detected as a younger object of the same class.
4. detect minimal deadlock chains starting from a call in $\mathcal{C}_b$ (if any).

If no deadlock chain is found, one may conclude that the program has no real deadlock, i.e., no runtime deadlock cycle.

Handling of subinterfaces. We allow subinterfaces, i.e., an interface may inherit methods from several interfaces, and each interface may have several class implementations. If a class C implements an interface I it also implements any superinterface of I. Thus a superinterface of I will in general have more class implementations than I. When we consider a call to method m of interface I we need to consider all further calls from implementations of m of classes implementing I (or a subinterface of I).

4 Deadlock analysis of Publisher-Subscriber

In this section we use the Publisher-Subscriber example to demonstrate our static analysis of possible deadlock chains. We will first show that the Publisher-Subscriber example given in Fig. 2 is deadlock-free using our approach. Thereafter, we will make non-trivial modifications to the example and show how deadlock chains can be detected.

For the example, we can statically detect that proxy is younger than self in class Service, as proxy can only be updated through proxy:= new Proxy (limit,self,prod) in line 18. Similarly, for class Proxy, we have that nextProxy is younger than self. However, in either class we cannot conclude that lastProxy is younger than self.

The program has the following blocking calls:

Service : subscribe → ProxyI : add – line 21
Proxy : add → Proxy : add – line 33

And the program has the following non-blocking calls:

Proxy : start_publish → ProducerI : detectNews – line 37

Note that in Proxy : add → Proxy : add, the callee is of class Proxy rather than interface ProxyI, due to our static detection of younger objects. According to step 2 above we may then ignore the call Proxy : add → Proxy : add. Thus a possible deadlock chain may only start from the call Service : subscribe → ProxyI : add. However there are no call chains leading back to ServiceI. Therefore there is no deadlock in the example.

Modifications of the example

In order to complicate analysis, we modify the code in Fig. 2 by replacing all trivial calls by blocking calls. This gives the following calls:

Service : init → self : produce
Service : produce → Proxy : start_publish
Service : subscribe → ProxyI : add
Proxy : start_publish → ProducerI : detectNews
Proxy : start_publish → self : publish
Proxy : add → Proxy : add – younger
Proxy : publish → ClientI : signal
Proxy : publish → ServiceI : produce
Proxy : publish → Proxy : publish – younger

We then remove calls to the same method of a younger object. In our case, the two calls of Proxy marked "younger" can be removed.

Static analysis detects a deadlock cycle, produce, start_publish, publish, produce, starting in class Service and ending in ServiceI. This represents a real deadlock. It can be avoided by making any of the three last calls trivial (as in the original code). This deadlock chain then disappears.

We also detect the deadlock cycle produce, start_publish, publish, signal, given that a Client has made the initial blocking produce call. This represents a real deadlock (if the client has made an add before the produce). This deadlock can be avoided by disallowing Clients to make produce calls. The cointerface mechanism of Creol makes this possible by stating that the produce method has ServiceI as cointerface, which means that the type-checking ensures that produce are only called from classes implementing ServiceI. Then type correctness implies that Clients cannot make such calls. Thus this cycle will not result from the static analysis.

Another way of removing the deadlock is to make the signal call a trivial call (as in the original code). This is sufficient to avoid the deadlock chain. No further deadlock chains exist (apart from variations of the ones above).

5 Related work

Most works on deadlock detection concern the thread-based model of Java where locks are used to get access to shared resources. Deadlock detection is then concerned with the usage of locks and signaling/notification. A survey is for instance given in the recent thesis [10], which develops a behavioral type and effect system to statically capture lock manipulation by processes for a thread and lock based concurrent language. The thesis addresses deadlocks by detecting cycles of processes waiting for shared locks.

In the present setting we consider a more high-level programming paradigm where the use of locks and signaling is avoided. Deadlocks are caused by call-cycles made up of blocking and non-blocking calls, and only blocking calls made by an object can cause that object to deadlock.

The paper [9] makes a discussion on how to guarantee freedom of deadlocks in the setting of concurrent objects with asynchronous methods and futures. The approach combines a notion of younger objects with a partial ordering of interfaces. The discussion is limited to classes implementing only one interface and to interfaces without inheritance of other interfaces. However, the future mechanism makes the identification of call cycles difficult. For each query on a future one would need to know the callee (i.e., the object executing the method producing the future). In order to solve this, one can make a static approximation of the possible callees of the future, considering (in the worst case) all classes of the program. Thus the analysis is not modular. In contrast our work allows classes implementing several interfaces, and interface inheritance is allowed. The problem related to detection of callees of futures is avoided since for each call the callee is given explicitly in the code. We here do not use a partial ordering of interfaces. However, a partial ordering of interfaces may be combined with our approach, taking care that whenever a subinterface I is ordered then any superinterface of I can be ordered in the same way, requiring that the resulting order should still be a partial ordering.

In [11] deadlock detection of Creol-like programs is done by transforming the program into a Petri Net. Futures are allowed, and the information associated with a future depends on the whole program. Compared to our approach, a drawback with this method is that it is not easy to do by hand; and it is not able to handle our main example, as (unbounded number of) new objects are not supported.

Deadlock analysis of ABS programs with futures is studied in [12]. A "challenging example" considered there can be formulated in our setting as

```
class Worker(Factory fc) { Void assignWork(Int n) { Void v;
                          if n>0 then v=fc.createWorker(n−1) fi }}
class Factory() { Void createWorker(Int n) { Void v; Worker w;
                          w = new Worker(this); v=await w!assignWork(n)}}
```

It is found deadlock free, with our as well as their approach, due to a freshness condition similar to ours. But reductions similar to *younger* objects (as in our main example) are not found. Their approach considers abstractions of the actual objects of closed systems, whereas our approach considers classes and interfaces and is suitable for open environments.

6 Conclusions

In this paper we have considered concurrent objects. This concurrency model is different from that found in mainstream languages such as Java. We find it interesting since it is based on high-level synchronization primitives, rather than locks and signaling, and it allows the caller to control the waiting time by means of different ways of calling a method, suspending, blocking, or non-blocking. In addition it directly supports distribution, autonomy, message-based communication, and object-orientation. Internal data structures are programmed by the use of data types.

The concurrency model has recently been the theme of several EU projects, including Credo, Hats, Envisage, and Upscale. Tool support for this concurrency model have been investigated in several ways, including compilation to more low-level languages including Java, Erlang, and Scala. These works show promising results when compared to other concurrency models for distributed systems based on imperative programming.

In this paper we look at the problem of static deadlock analysis for the concurrency model, considering a high-level kernel language supporting the different calling mechanisms. The paper shows that by means of a relatively simple analysis one can effectively detect possible deadlocks in non-trivial examples, as demonstrated by looking at variations over a main example. Language constructs not relevant for deadlock problems have been abstracted away, and we have deliberately avoided futures. Thus the results of this paper seem to confirm the claim that the considered concurrency model is appealing for modeling and designing distributed systems. In particular the presented deadlock detection analysis is simple to do by hand, and is also simple to automate. As future work we would like to formally prove the soundness of our analysis method, and experiment with a tool implementation based on the Creol tool chain. And we would like to extend the language and analysis to support method delegation, which offers some of the flexibility of futures but seems simpler to analyze.

References

[1] E. G. Coffman, M. Elphick, A. Shoshani, System deadlocks, ACM Comput. Surv. 3 (2) (1971) 67–78.

[2] J. Armstrong, Erlang, Commun. ACM 53 (9) (2010) 68–75.

[3] M. Odersky, L. Spoon, B. Venners, Programming in Scala. A comprehensive step-by-step guide, Artima Developer, 2008.

[4] E. Giachino, C. Laneve, Analysis of deadlocks in object groups., in: R. Bruni, J. Dingel (Eds.), FMOODS/FORTE, Vol. 6722 of Lecture Notes in Computer Science, Springer, 2011, pp. 168–182.

[5] E. Kerfoot, S. McKeever, F. Torshizi, Deadlock freedom through object ownership, in: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, IWACO '09, ACM, New York, NY, USA, 2009, pp. 3:1–3:8. doi:10.1145/1562154.1562157.

[6] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACMTransactions on Programming Languages and Systems 23 (3) (2001) 396–450.

[7] 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.

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

[9] C. C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, Journal of Logical and Algebraic Methods in Programming (To appear) (0) (2014) –. doi:http://dx.doi.org/10.1016/j.jlamp.2014.03.003.

[10] K. I. Pun, Behavioural static analysis for deadlock detection, Ph.D. thesis, Department of informatics, University of Oslo, Norway (2014).

[11] F. S. de Boer, M. Bravetti, I. Grabe, M. D. Lee, M. Steffen, G. Zavattaro, A Petri Net based analysis of deadlocks for active objects and futures., in: C. S. Pasareanu, G. Salaün (Eds.), FACS, Vol. 7684 of Lecture Notes in Computer Science, Springer, 2012, pp. 110–127.

[12] A. Flores-Montoya, E. Albert, S. Genaim, May-happen-in-parallel based deadlock analysis for concurrent objects, in: D. Beyer, M. Boreale (Eds.), Formal Techniques for Distributed Systems, Vol. 7892 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 273–288.