Progress of Concurrent Objects

Other titles in Foundations and Trends® in Programming Languages

Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation
Antoine Miné
ISBN: 978-1-68083-386-7

Program Synthesis
Sumit Gulwani, Oleksandr Polozov and Rishabh Singh
ISBN: 978-1-68083-292-1

Programming with Big Code
Martin Vechev and Eran Yahav
ISBN: 978-1-68083-230-3

Behavioral Types in Programming Languages
Davide Ancona et al.
ISBN: 978-1-68083-134-4

Computer-Assisted Query Formulation
Alvin Cheung and Armando Solar-Lezama
ISBN: 978-1-68083-036-1

Progress of Concurrent Objects

Hongjin Liang
State Key Laboratory for Novel Software Technology
Nanjing University
China
hongjin@nju.edu.cn

Xinyu Feng
State Key Laboratory for Novel Software Technology
Nanjing University
China
xyfeng@nju.edu.cn

now
the essence of knowledge
Boston — Delft

Foundations and Trends® in Programming Languages

Published, sold and distributed by:
now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:
now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

H. Liang and X. Feng. *Progress of Concurrent Objects*. Foundations and Trends® in Programming Languages, vol. 5, no. 4, pp. 282–414, 2020.

ISBN: 978-1-68083-673-8

Foundations and Trends® in Programming Languages
Volume 5, Issue 4, 2020

Editorial Board

Editorial Scope

Topics

Foundations and Trends® in Programming Languages publishes survey and tutorial articles in the following topics:

- Abstract Interpretation
- Compilation and Interpretation Techniques
- Domain Specific Languages
- Formal Semantics, including Lambda Calculi, Process Calculi, and Process Algebra
- Language Paradigms
- Mechanical Proof Checking
- Memory Management
- Partial Evaluation
- Program Logic
- Programming Language Implementation
- Programming Language Security
- Programming Languages for Concurrency
- Programming Languages for Parallelism
- Program Synthesis
- Program Transformations and Optimizations
- Program Verification
- Runtime Techniques for Programming Languages
- Software Model Checking
- Static and Dynamic Program Analysis
- Type Theory and Type Systems

Information for Librarians

Foundations and Trends® in Programming Languages, 2020, Volume 5, 4 issues. ISSN paper version 2325-1107. ISSN online version 2325-1131. Also available as a combined paper and online subscription.

Contents

1 Introduction **3**
1.1 General Motivation 5
1.2 Overview 8

2 Background **10**
2.1 Linearizability 10
2.2 Progress Properties 12
2.3 Contextual Refinement and Abstraction Theorems 16
2.4 Verifying Progress Properties 22

3 Basic Technical Settings **32**
3.1 The Language 32
3.2 Execution Traces and Fairness of Scheduling 38

4 Linearizability and Contextual Refinement **41**
4.1 Linearizability 41
4.2 Contextual Refinement and Abstraction 43

5 Progress Properties **44**
5.1 Progress for Objects with Total Methods Only 44
5.2 Progress for Objects with Partial Methods 46

6 Progress-Aware Abstraction **50**
6.1 Overview of Our Results 50
6.2 Formalizing Progress-Aware Contextual Refinements 53
6.3 Abstraction for Wait-Free and Lock-Free Objects 56
6.4 Abstraction for Starvation-Free and Deadlock-Free Objects 60
6.5 Abstraction for PSF and PDF Objects 61

7 Verifying Progress of Concurrent Objects **68**
7.1 Challenges and Key Ideas 68
7.2 The Program Logic LiLi 75
7.3 Soundness 104
7.4 Examples 106

8 Related Work **116**
8.1 Progress Properties and Abstraction 116
8.2 Verification 117
8.3 Comparison with TaDA-Live 120

9 Conclusion and Future Work **125**

Acknowledgements **128**

References **129**

Progress of Concurrent Objects

Hongjin Liang[1] and Xinyu Feng[2]

[1]*State Key Laboratory for Novel Software Technology, Nanjing University, China; hongjin@nju.edu.cn*
[2]*State Key Laboratory for Novel Software Technology, Nanjing University, China; xyfeng@nju.edu.cn*

ABSTRACT

Implementations of concurrent objects should guarantee linearizability and a progress property such as wait-freedom, lock-freedom, starvation-freedom, or deadlock-freedom. These progress properties describe conditions under which a method call is guaranteed to complete. However, they fail to describe how clients are affected, making it difficult to utilize them in layered and modular program verification. Also we lack verification techniques for starvation-free or deadlock-free objects. They are challenging to verify because the fairness assumption introduces complicated interdependencies among progress of threads. Even worse, none of the existing results applies to concurrent objects with partial methods, i.e., methods that are supposed *not* to return under certain circumstances. A typical example is the `lock_acquire` method, which must *not* return when the lock has already been acquired.

In this tutorial we examine the progress properties of concurrent objects. We formulate each progress property (together with linearizability as a basic correctness requirement) in terms of contextual refinement. This essentially gives us progress-aware abstraction for concurrent objects. Thus,

Hongjin Liang and Xinyu Feng (2020), "Progress of Concurrent Objects", Foundations and Trends® in Programming Languages: Vol. 5, No. 4, pp 282–414. DOI: 10.1561/2500000041.

when verifying clients of the objects, we can soundly replace the concrete object implementations with their abstractions, achieving modular verification. For concurrent objects with partial methods, we formulate two new progress properties, partial starvation-freedom (PSF) and partial deadlock-freedom (PDF). We also design four patterns to write abstractions for PSF or PDF objects under strongly or weakly fair scheduling, so that these objects contextually refine their abstractions. Finally, we introduce a rely-guarantee style program logic LiLi for verifying linearizability and progress *together* for concurrent objects. It unifies thread-modular reasoning about all the six progress properties (wait-freedom, lock-freedom, starvation-freedom, deadlock-freedom, PSF and PDF) in one framework. We have successfully applied LiLi to verify starvation-freedom or deadlock-freedom of representative algorithms such as lock-coupling lists, optimistic lists and lazy lists, and PSF or PDF of lock algorithms.

1

Introduction

A concurrent object consists of shared data and a set of methods which provide an interface for client threads to access the shared data. Linearizability (Herlihy and Wing, 1990) has been used as a standard definition of the correctness of concurrent object implementations. It describes safety and functionality, but has no requirement about termination of object methods. Various progress properties, such as wait-freedom, lock-freedom, starvation-freedom and deadlock-freedom, have been proposed to specify termination of object methods. In their textbook Herlihy and Shavit (2008) give a systematic introduction of these properties.

Although program termination has been an obvious notion for sequential programs, it becomes much more complex in a concurrent setting. Termination of a method call in a thread is affected not only by the sequential behavior of the method code, but also by interference from the environment. Different implementations of the concurrent object methods have different tolerance of the interference. That's why we need these different progress properties.

We give two implementations of a simple counter object in Figure 1.1. The variable `x` (line `0`) is the object data implementing the counter. Figure 1.1(a) and (b) are two different implementations of the `inc`

```
0  int x; //object data
1  inc(){
2    local t, done := false;
3    while(!done){
4      t := x;
5      done := cas(&x, t, t+1);
6    }
7  }
```

(a)

```
8  inc(){
9    lock();
10     x := x+1;
11   unlock();
12 }
```

(b)

Figure 1.1: Implementations of the counter object.

method, which increments the counter. Figure 1.1(a) shows an optimistic implementation. It takes a snapshot `t` of the counter (line 4). The *atomic* **c**ompare-**a**nd-**s**wap (**cas**) command (line 5) compares the current value of `x` with `t`. If they are equal, it atomically sets `x` to `t+1` and returns `true`. Otherwise it does nothing and returns `false`, and the method has to run another round of the loop to roll back and retry the process. Figure 1.1(b) is a lock-based implementation, where the update of the shared variable `x` is protected by a lock. Here we omit the implementation of locks, which will be discussed later.

The different implementations of the `inc` method have different progress properties. We can consider the following client program to see their difference. The formal definitions of progress properties will be discussed later in Section 5.

```
inc()  ||  while(true){ inc(); }
```

If we use the optimistic version in Figure 1.1(a), the call of `inc()` in the left thread may never terminates because the **cas** command at line 5 may always fail due to the infinite number of calls of `inc()` in the right thread. However, whenever we suspend the execution of the right thread, the `inc()` in the left eventually terminates. Therefore we call Figure 1.1(a) a *non-blocking* implementation. Also, since at least one of the call of `inc()` in the whole program terminates, this is a *lock-free* implementation.

If we use the lock-based `inc` in Figure 1.1(b), whether the call of `inc()` in the left thread terminates or not depends on the implementation of the lock. If the lock implementation is fair, the `inc()` on the left always terminates, otherwise it may always fail to acquire the lock and may never terminate. In both cases, if we suspend the right thread when it is executing line `10`, the `inc()` on the left cannot terminate because it can never acquire the lock (which has been taken by the suspended right thread). So we say the lock-based implementation of `inc` is *blocking*. The termination of a method call relies on both the lock algorithm and the fairness of scheduling.

The goal of this tutorial is to help the reader understand the various progress properties of concurrent objects. We formulate each progress property (together with linearizability as a basic correctness requirement) in terms of contextual refinement. This essentially gives us progress-aware abstraction for concurrent objects. We also introduce a program logic LiLi to formally verify progress properties.

1.1 General Motivation

1.1.1 Progress-Aware Abstraction

Progress properties describe conditions under which method calls are guaranteed to successfully complete in an execution. For example, lock-freedom guarantees that "infinitely often some method call finishes in a finite number of steps" (Herlihy and Shavit, 2008). They are difficult to use in a modular and layered program verification because they fail to describe how the progress properties affect clients.

In a modular verification of client threads, the concrete implementation Π of the object methods should be replaced by an abstraction (or specification) Π' that consists of equivalent methods. The progress properties should then characterize whether and how the behaviors of a client program will be affected if a client uses Π instead of Π'. In particular, we are interested in systematically studying whether the termination of a client using the abstract methods Π' will be preserved when using an implementation Π with some progress guarantee.

Previous work on verifying the *safety* of concurrent objects (e.g., Filipović *et al.*, 2009; Liang and Feng, 2013) has shown that linearizability—a standard safety criterion for concurrent objects—and contextual refinement are equivalent. Informally, an implementation Π is a contextual refinement of a (more abstract) implementation Π', if every observable behavior of any client program using Π can also be observed when the client uses Π' instead. To obtain equivalence to linearizability, the observable behaviors include I/O events but not divergence (i.e., non-termination). Recently, Gotsman and Yang (2011) showed that a client program that diverges using a linearizable and *lock-free* object must also diverge when using the abstract operations instead. Their work reveals a connection between lock-freedom and a form of contextual refinement which preserves termination as well as safety properties. It is unclear how other progress guarantees affect termination of client programs and how they are related to contextual refinements.

This tutorial studies four commonly used progress properties (wait-freedom, lock-freedom, starvation-freedom and deadlock-freedom) and their relationships to contextual refinements. We show that, when progress properties are taken into account, one may have the corresponding progress-aware contextual refinement to reestablish the equivalence. We give different abstract specifications Π' for different progress properties. The equivalence results allow us to build abstractions for linearizable objects so that safety and progress of the client code can be reasoned about at a more abstract level.

1.1.2 Program Logic for Progress Verification

Recent program logics for verifying concurrent objects either prove only linearizability and ignore the issue of termination (e.g., Derrick *et al.*, 2011; Liang and Feng, 2013; Turon *et al.*, 2013a; Vafeiadis, 2008), or aim for non-blocking progress properties (e.g., da Rocha Pinto *et al.*, 2016; Gotsman *et al.*, 2009; Hoffmann *et al.*, 2013; Liang *et al.*, 2014), which cannot be applied to blocking algorithms that progress only under fair scheduling. The fairness assumption introduces complicated interdependencies among progress properties of threads, making it incredibly more challenging to verify the lock-based algorithms than

their non-blocking counterparts. We will explain the challenges in detail in Subsection 7.1.

It is important to note that, although there has been much work on deadlock detection or deadlock-freedom verification (e.g., Boyapati *et al.*, 2002; Leino *et al.*, 2010; Williams *et al.*, 2005), deadlock-freedom is often defined as a safety property, which ensures the lack of circular waiting for locks. It does not prevent live-lock or non-termination inside the critical section. Another limitation of this kind of work is that it often assumes built-in lock primitives, and lacks support of ad-hoc synchronization (e.g., mutual exclusion achieved using spin-locks implemented by the programmers). The deadlock-freedom we discuss in this tutorial is a liveness property and its definition does not rely on built-in lock primitives. We discuss more related work on liveness verification in Section 8.

In this tutorial we introduce LiLi, a new rely-guarantee style logic for concurrent objects. It unifies verification of linearizability, wait-freedom, lock-freedom, starvation-freedom and deadlock-freedom in one framework (the name LiLi stands for Linearizability and Liveness). In particular, it supports verification of both mutex-based pessimistic algorithms (including fine-grained ones such as lock-coupling lists) and optimistic ones such as optimistic lists and lazy lists. The unified approach allows us to prove *in the same logic*, for instance, the lock-coupling list algorithm is starvation-free if we use fair locks, e.g., ticket locks (Mellor-Crummey and Scott, 1991), and is deadlock-free if regular test-and-set based spin locks (Herlihy and Shavit, 2008) are used.

1.1.3 Concurrent Objects with Partial Methods

However, *none* of the aforementioned progress-related results applies to concurrent objects with partial methods! A method is *partial* if it is supposed *not* to return under certain circumstances. A typical example is the `lock_acquire` method, which must *not* return when the lock has already been acquired. Concurrent objects with partial methods simply do not satisfy any of the aforementioned progress properties, and people do not know how to give progress-aware abstract specifications for them either. The existing studies on progress properties and progress-aware

contextual refinements have been limited to concurrent objects with total specifications.

As an awkward consequence, we cannot treat lock implementations as objects when we study progress of concurrent objects. Instead, we have to treat `lock_acquire` and `lock_release` as *internal* functions of other lock-based objects. Also, without a proper progress-aware abstraction for locks, we have to redo the verification of `lock_acquire` and `lock_release` when they are used in different contexts (Liang and Feng, 2016), which makes the verification process complex and painful. Note that locks are not the only objects with partial methods. Concurrent sets, stacks and queues may also have methods that intend to block. For instance, it may be sensible for a thread attempting to pop from an empty stack to block, waiting until another thread pushes an item. The reasoning about these objects suffers from the same problems too when progress is concerned.

In this tutorial, we specify and verify progress of concurrent objects with partial methods. We define partial starvation-freedom (PSF) and partial deadlock-freedom (PDF) as progress properties for objects with partial methods, and design abstraction patterns under strongly and weakly fair scheduling. We prove that given a linearizable object implementation Π with partial methods, the contextual refinement between Π and its abstraction Π' under a certain kind of fair scheduling is equivalent to PSF/PDF of Π. We also extend the program logic LiLi to support partial specifications and to reason about blocking primitives. It verifies the contextual refinement between Π and Π', which ensures linearizability and the progress property of Π.

1.2 Overview

The goal of this tutorial is to help the reader understand the various progress properties of concurrent objects.

We start with an informal overview of the background in Section 2. We informally describe the traditional four progress properties (wait-freedom, lock-freedom, starvation-freedom and deadlock-freedom), and analyze the challenges in supporting objects with partial methods.

In Section 3, we introduce the basic technical settings. We define a simple object language, and the generation of execution traces from the operational semantics. We also define fairness of scheduling over the traces.

In Section 4, we define linearizability and the basic contextual refinement which is equivalent to linearizability.

In Section 5, we formulate the four traditional progress properties and the two new progress properties for objects with partial methods.

In Section 6, we give the progress-aware contextual refinement and the abstraction theorems.

In Section 7, we present the program logic LiLi and show the examples we have verified.

Finally, we discuss related work in Section 8 and conclude in Section 9.

References

Abadi, M. and L. Lamport (1995). "Conjoining specifications". *ACM Trans. Program. Lang. Syst.* 17(3): 507–535.

Aspnes, J. and M. Herlihy (1990). "Wait-free data structures in the asynchronous PRAM model". In: *SPAA*. 340–349.

Back, R. and Q. Xu (1998). "Refinement of fair action systems". *Acta Inf.* 35(2): 131–165.

Boström, P. and P. Müller (2015). "Modular verification of finite blocking in non-terminating programs". In: *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP 2015)*. 639–663.

Boyapati, C., R. Lee, and M. Rinard (2002). "Ownership types for safe programming: Preventing data races and deadlocks". In: *OOPSLA*. 211–230.

Calcagno, C., M. J. Parkinson, and V. Vafeiadis (2007). "Modular safety checking for fine-grained concurrency". In: *Proceedings of the 14th International Symposium on Static Analysis (SAS 2007)*. 233–248.

Cook, B., A. Podelski, and A. Rybalchenko (2011). "Proving program termination". *Commun. ACM*. 54(5): 88–98.

da Rocha Pinto, P., T. Dinsdale-Young, P. Gardner, and J. Sutherland (2016). "Modular termination verification for non-blocking concurrency". In: *Proceedings of the 25th European Symposium on Programming Languages and Systems (ESOP 2016)*. 176–201.

Derrick, J., G. Schellhorn, and H. Wehrheim (2011). "Mechanically verified proof obligations for linearizability". *ACM Trans. Program. Lang. Syst.* 33(1): 4:1–4:43.

Doherty, S., L. Groves, V. Luchangco, and M. Moir (2004). "Formal verification of a practical lock-free queue algorithm". In: *FORTE*. 97–114.

Dongol, B. (2006). "Formalising progress properties of non-blocking programs". In: *ICFEM*. 284–303.

D'Osualdo, E., A. Farzan, P. Gardner, and J. Sutherland (2019). "TaDA live: Compositional reasoning for termination of fine-grained concurrent programs". arXiv: 1901.05750.

Feng, X. (2009). "Local rely-guarantee reasoning". In: *POPL*. 315–327.

Filipović, I., P. O'Hearn, N. Rinetzky, and H. Yang (2009). "Abstraction for concurrent objects". In: *Proceedings of the 18th European Symposium on Programming (ESOP 2009)*. 252–266.

Fossati, L., K. Honda, and N. Yoshida (2012). "Intensional and extensional characterisation of global progress in the π-calculus". In: *CONCUR*. 287–301.

Gotsman, A., B. Cook, M. J. Parkinson, and V. Vafeiadis (2009). "Proving that non-blocking algorithms don't block". In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*. 16–28.

Gotsman, A. and H. Yang (2011). "Liveness-preserving atomicity abstraction". In: *Proceedings of the 38th International Conference on Automata, Languages and Programming (ICALP 2011)*. 453–465.

Gotsman, A. and H. Yang (2012). "Linearizability with ownership transfer". In: *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR 2012)*. 256–271.

Gu, R., Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo (2016). "CertiKOS: An extensible architecture for building certified concurrent OS kernels". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI 2016)*. 653–669.

Harris, T. L. (2001). "A pragmatic implementation of non-blocking linked-lists". In: *DISC*. 300–314.

Heller, S., M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit (2005). "A lazy concurrent list-based set algorithm". In: *OPODIS*. 3–16.

Hendler, D., N. Shavit, and L. Yerushalmi (2004). "A scalable lock-free stack algorithm". In: *SPAA*. 206–215.

Henzinger, T. A., O. Kupferman, and S. K. Rajamani (2002). "Fair simulation". *Inf. Comput.* 173(1): 64–81.

Herlihy, M. and N. Shavit (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann.

Herlihy, M. and N. Shavit (2011). "On the nature of progress". In: *Proceedings of the 15th International Conference on Principles of Distributed Systems (OPODIS 2011)*. 313–328.

Herlihy, M. and J. Wing (1990). "Linearizability: A correctness condition for concurrent objects". *ACM Trans. Program. Lang. Syst.* 12(3): 463–492.

Hoffmann, J., M. Marmar, and Z. Shao (2013). "Quantitative reasoning for proving lock-freedom". In: *Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2013)*. 124–133.

Jacobs, B., D. Bosnacki, and R. Kuiper (2015). "Modular termination verification". In: *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP 2015)*. 664–688.

Jones, C. B. (1983). "Tentative steps toward a development method for interfering programs". *ACM Trans. Program. Lang. Syst.* 5(4): 596–619.

Jung, R., D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer (2015). "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*. 637–650.

Khyzha, A., M. Dodds, A. Gotsman, and M. J. Parkinson (2017). "Proving linearizability using partial orders". In: *Proceedings of the 26th European Symposium on Programming (ESOP 2017)*. 639–667.

Leino, K. R. M. and P. Müller (2009). "A basis for verifying multi-threaded programs". In: *ESOP*. 378–393.

Leino, K. R. M., P. Müller, and J. Smans (2010). "Deadlock-free channels and locks". In: *ESOP*. 407–426.

Liang, H. and X. Feng (2013). "Modular verification of linearizability with non-fixed linearization points". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. 459–470.

Liang, H. and X. Feng (2016). "A program logic for concurrent objects under fair scheduling". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. 385–399.

Liang, H. and X. Feng (2018a). "Progress of concurrent objects with partial methods". *Proc. ACM Program. Lang.* 2(POPL): Article 20.

Liang, H. and X. Feng (2018b). "Progress of concurrent objects with partial methods (extended version)". *Tech. Rep.* https://cs.nju.edu.cn/hongjin/papers/popl18-partial-tr.pdf.

Liang, H., X. Feng, and Z. Shao (2014). "Compositional verification of termination-preserving refinement of concurrent programs". In: *Proceedings of the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS 2014)*. Article 65.

Liang, H., J. Hoffmann, X. Feng, and Z. Shao (2013). "Characterizing progress properties of concurrent objects via contextual refinements". In: *Proceedings of the 24th International Conference on Concurrency Theory (CONCUR 2013)*. 227–241.

Mellor-Crummey, J. M. and M. L. Scott (1991). "Algorithms for scalable synchronization on shared-memory multiprocessors". *ACM Trans. Comput. Syst.* 9(1): 21–65.

Michael, M. M. (2002). "High performance dynamic lock-free hash tables and list-based sets". In: *SPAA*. 73–82.

Michael, M. M. and M. L. Scott (1996). "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms". In: *PODC*. 267–275.

Parkinson, M., R. Bornat, and C. Calcagno (2006). "Variables as resource in Hoare logics". In: *LICS*. 137–146.

Petrank, E., M. Musuvathi, and B. Steensgaard (2009). "Progress guarantee for parallel programs via bounded lock-freedom". In: *PLDI*. 144–154.

Schellhorn, G., O. Travkin, and H. Wehrheim (2016). "Towards a thread-local proof technique for starvation freedom". In: *Proceedings of the 12th International Conference on Integrated Formal Methods (IFM 2016)*. 193–209.

Stark, E. W. (1985). "A proof technique for rely/guarantee properties". In: *FSTTCS*. 369–391.

Stølen, K. (1992). "Shared-state design modulo weak and strong process fairness". In: *FORTE*. 479–498.

Tassarotti, J., R. Jung, and R. Harper (2017). "A higher-order logic for concurrent termination-preserving refinement". In: *Proceedings of the 26th European Symposium on Programming (ESOP 2017)*. 909–936.

Treiber, R. K. (1986). "System programming: Coping with parallelism". *Tech. Rep.* RJ 5118. IBM Almaden Research Center.

Turon, A., D. Dreyer, and L. Birkedal (2013a). "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency". In: *ICFP*. 377–390.

Turon, A., J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer (2013b). "Logical relations for fine-grained concurrency". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013)*. 343–356.

Vafeiadis, V. (2008). "Modular fine-grained concurrency verification". *Tech. Rep.* PhD Thesis.

Williams, A., W. Thies, and M. D. Ernst (2005). "Static deadlock detection for java libraries". In: *ECOOP*. 602–629.

Xu, Q., W. P. de Roever, and J. He (1997). "The rely-guarantee method for verifying shared variable concurrent programs". *Formal Asp. Comput.* 9(2): 149–174.