JDala - A Simple Capability System for Java

Quinten Smit*, Jens Dietrich*, Michael Homer*, Andrew Fawcet*, James Noble†
*Victoria University of Wellington, School of Engineering and Computer Science, Wellington, New Zealand
smitquin@myvuw.ac.nz,{jens.dietrich,michael.homer,andrew.fawcet}@vuw.ac.nz
†Creative Research and Programming, Wellington, New Zealand, kjx@acm.org

***Abstract*—Dala is a novel capability-based programming model that ensures data-race freedom while also supporting efficient inter-thread communication. While Dala has been designed to inform the design of future programming languages, the question arises whether existing languages can be retrofitted with Dala capabilities. We report such a design called JDala. In JDala, Dala capabilities are added to Java using annotations and interpreted using bytecode instrumentation. With some examples we demonstrate that by adding three simple annotations to the language, we can avoid concurrency bugs like deadlocks and unexpected program behaviour resulting from shallow immutability of Java standard library APIs.**

JDala demo: https://youtu.be/QddK1q35h-U

***Index Terms*—object capabilities, java, instrumentation, concurrency, immutability**

I. Introduction

Concurrent programming is ubiquitous today, driven by hardware advancements and software demands, but prone to subtle errors that often pass undetected and cause unexpected behaviour [28], [30], [37]. Mainstream programming languages—especially those with shared-memory concurrency—leave developers vulnerable to concurrency bugs such as data races, race conditions, and deadlocks, which are notoriously difficult to detect, reproduce, and debug. Dala [14], [38] is a simple, novel model of object-oriented capabilities with dynamic enforcement. Dala allows parts of a program's state to be protected from mutation (`imm`), concurrent access across threads (`local`), or aliased references (`iso`), thereby helping inhibit data races and related errors. These three capability flavours enforce constraints: immutability prohibits state change, isolation ensures thread exclusivity and non-aliasing, and locality guarantees thread affinity. Violations of these constraints are detected dynamically and halted with informative errors, enabling a fail-fast programming style [39] that improves safety and ability to debug. Work on Dala has been based on extensions to the Grace programming language [1], a simple, educational language designed with minimal concepts. This prior work has not addressed how the model might be applied to existing, complex, production-oriented languages with real-world concurrency requirements. JDala is an attempt to augment Java with Dala capabilities: to allow Java objects to be given one of the Dala flavours, and to detect and report violations of their constraints at run time. Unlike static approaches such as Rust's affine type system or Pony's actor-based isolation, JDala uses lightweight annotations and bytecode instrumentation to retroactively enforce capability restrictions within standard Java programs. With only three core annotations, Java developers can sidestep common concurrency pitfalls—such as unsafe aliasing or illegal thread-sharing—while retaining the structure and familiarity of the Java ecosystem. JDala lets the program execute until a visible error occurs, then prevents the effects of that error from propagating further. This fail-fast behaviour assists developers in debugging and prevents concurrency bugs, like race conditions or misuse of immutability, from corrupting the run-time state of the program.

This work is supported in part by the Royal Society of New Zealand Te Apārangi Marsden Fund Te Pūtea Rangahau a Marsden grant CRP2101.

An alternative to implement Dala in Java would have been the use of a pluggable types system like *JavaCOP* [32] or the *checkerframework* [13]. In particular for the *checkerframework*, several checkers similar to Dala (e.g. immutability) already exist. Rules are enforced statically, i.e. at compile time. A drawback is the handling of dynamic programming patterns like reflection that are pervasive in Java [41] and notorious difficult to model by static analyses [12], [29], resulting in unsoundness or false alerts (if the analysis over-approximates dynamic language features, e.g. by considering all objects returned by `Method::invoke` as immutable). JDala's dynamic approach can avoid this, although at the cost of runtime overhead.

The contributions of this paper are: (1) A design for JDala, adding Dala capabilities to Java. (2) A prototype implementation of JDala using annotations and bytecode instrumentation. (3) Examples of how JDala can avoid common bugs.

II. Related Work

Concurrent programming is hard [26], [33] because concurrent programs do two or more things at the same time. Even if one thread of execution is perfectly correct, another thread can interfere with its behaviour corrupting data, triggering deadlock, or crashing the whole system [15]. The most important kind of interference between threads are *data races* — when one thread writes to shared memory that another thread is reading or writing. The problem with data races is that values stored into memory, and the values returned by any reads, may be essentially arbitrary, depending on the semantics of the particular language, its implementation, and even the underlying CPU and memory architecture. This is exacerbated because concurrent programs are typically written in low-level languages such as C, C++, BLISS-32, or Swift,

which provide few correctness guarantees, making it easy to write **wrong** concurrent programs [19], [30].

A range of programming techniques has been developed avoid these problems [26], [33], aiming to provide *safe concurrency* where data races are prevented either by the design of the programming language itself or its associated tools. Unfortunately, while concurrent programming is hard, safe concurrent programming language design turns out to be even harder [3], [23].

While full-scale proof systems [10], [11], [20], [22], [27], [42], [43] can guarantee correctness for programs in almost any languages, they are too specialised for developers to use in practice [16], [25].

Contemporary static programming languages such as Rust [23], Pony [6], Encore [3], Obsidian [7], and Verona [4] have demonstrated the efficacy of *static ownership* [5], [21] to ensure concurrent programs are safe: Rust in particular has been adopted by Microsoft [18], [24]. By keeping track of each object's ownership, these languages can determine when an object may be used, when it may be changed, and the effects those changes can have on the rest of the program.

On the other hand, *dynamically* concurrent languages like JavaScript [34], E [36], and AmbientTalk [9] support simple, concurrency-safe programming by design, ruling out whole classes of bugs. Programmers don't need to annotate their code (an input stream could be just declared as "`in`") making programs easier to read and write. Unfortunately, these kinds of dynamic approaches achieve safety and simplicity by sacrificing performance, in particular by banning efficient communication between concurrent threads.

In contemporary industrial software development, a range of techniques for dynamic race detection are used in practice across a range of programming languages [17]. Go, for example, which includes syntax and implementation support designed to encourage the use of many lightweight threads, recommends the use of a race detector [8]. The issue with race detectors is that they detect (potential) data races — they don't explain why a particular data race occurs.

In many ways the approach closest to Dala is a concurrent proposal for Python, also based on dynamic ownership checking to maintain an explicit relationship between objects and the threads that can access them [40]. This proposal, *Lungfish*, organises objects in regions based on their ownership. Lungfish supports equivalents of Dala's immutable, isolated, and shared objects, along with a fourth category, "cowns" (pronounced "cones") for objects guarded by a lock.

III. Motivating Examples

A. Immutable Sorted Lists

Consider the following code in Listing 1 [1]. This code attempts to create an unmodifiable sorted list of `Person` instances. However, while `Collections::unmodifiableList` makes the list immutable, the objects within the list can still be mutated, including changes to the name property used as a sort key. Once such mutations have taken place, it is no longer guaranteed that the members of the list are sorted by name, and an application that incorrectly relies on such an assumption may exhibit unexpected behaviour.

```
1 List<Person> people =  .. ;
2 Collections.sort(people,
3 Comparator.comparing(Person::getName));
4 Collections.unmodifiableList(people);
5 for (Person p:people)  System.out.println(p);
```

Listing 1. Erroneous Attempt to Make a Sorted List Immutable

In JDala, this can be prevented by annotating the respective list as `@Immutable`. This is shown in Listing 2, line 4. In contrast to `Collections::unmodifiableList`, immutability is now deep, i.e. it also applies to all objects within the list. This is enforced dynamically (i.e. at runtime) by intercepting attempts to change the state of objects in the list. Attempts to change the state of immutable objects are signalled with a `DalaCapabilityViolationException`.

```
1 List<Person> people =  .. ;
2 Collections.sort(people,
3 Comparator.comparing(Person::getName));
4 @Immutable immutPeople=people;
5 ..
```

Listing 2. Make a Sorted List Immutable with JDala

In the sorted list example, calls to `Person::setName` (which writes to the `Person::name` field) will now result in a runtime exception. This is *fail-fast behaviour* [39], i.e. unexpected behaviour is avoided by producing an informative error signal at the point where the issue occurs.

B. Deadlock Prevention

Consider Listing 3. This is a simple method for transferring money between two accounts. To ensure that sufficient funds are available, the respective accounts are locked using the `synchronized` keyword. However, if an application encounters a situation in which money has to be transferred within a short time window between two accounts in both directions, a deadlock can occur causing the application to stall [2].

```
1  void transfer(Account from, Account to, double amount) {
2    synchronized (from) {
3      from.withdraw(amount);
4      Thread.sleep(1_000); // to simulate database write(s)
5      synchronized (to) to.deposit(amount);
6  }}
7  ..
8  ThreadPoolExecutor tpool = .. ;
9  @Local Account acc1,acc2  ..;
10 Future f1 = tpool.submit(() -> transfer(acc1, acc2, 50));
11 Future f2 = tpool.submit(() -> transfer(acc2, acc1, 80));
```

Listing 3. Money transfer implementation prone to deadlock

[1]The full source code of the examples used here can be found in the project repository: https://github.com/jensdietrich/jdala/blob/main/jdala-core/src/test/java/nz/ac/wgtn/ecs/jdala/realWorldExamples/. Examples are written as unit tests with oracles illustrating their behaviour with and without JDala instrumentation.

[2]In the full example code, this is illustrated by using a timeout oracle. The deadlock can be observed with a JMX client like VisualVM.

By running this program with the JDala agent and annotating the accounts with `@Local`, the deadlock can be prevented. The instrumented program immediately fails with a JDala exception (wrapped in the `ExecutionException` referenced by the future instances) indicating that the `Account` instances are associated with the main thread, but used in different (thread pool) threads.

IV. The Design of JDala

A. Capabilities as Annotations

In JDala, the Dala capabilities `imm`, `iso` and `local` are represented using the three annotation types `@Immutable`, `@Isolated` and `@Local`, respectively. Global data structures (maps that are static members of the `JDala` class) are used to track objects with the respective capabilities.

Since in Java objects cannot be annotated directly, the association between objects and object capabilities is achieved by annotating local variables pointing to objects. This is illustrated in Listing 4. In line 1, the newly created `List` instance is marked as immutable. A second list created in line 2 is not marked as immutable at the allocation site, but later. This is achieved by an assignment to the annotated variable `list3`.

```
1 @Immutable List list = new ArrayList();
2 List list2 = new ArrayList();
3 @Immutable List list3 = list2;
```

Listing 4. Associating objects with capabilities

B. Enforcing the Semantics of Capabilities

The semantics of the respective capabilities are implemented using instrumentation that injects code to enforce them. JDala uses ASM [2] for this purpose, and the project builds an agent that can then be attached to any Java application either statically (using the `-javaagent` argument) or dynamically. The instrumentation controls the heap of an instrumented application by maintaining safe objects [3] in global data structures, and intercepting and checking bytecode instructions reading and writing fields (i.e. traversing and manipulating the heap). Instructions for reference type fields, special instructions for fields with one of the various Java primitive types, and instructions to access array slots are all instrumented.

The injected code broadly falls into two categories: *registration* and *enforcement*. When annotated references to objects are encountered, the corresponding objects are registered as immutable, local or isolated in static maps maintained by the `JDala` class. In case of immutability, referenced objects are registered as immutable as well. For this purpose, a simple reflection-based heap traversal is performed on the object to be registered as immutable. If already registered objects are re-registered with a weaker capability [4], a `DalaRestrictionException` is raised. This functionality is implemented in various `JDala::register*` methods.

When fields of registered objects are accessed, the injected code invokes check methods `JDala::validate*` to enforce the capability contract. Violations are signalled by raising a `DalaCapabilityViolationException` exception.

C. Object vs Class-Based Capabilities

JDala provides two mechanisms to define the capabilities. The primary method uses the object-level annotations discussed in Section IV-A. This method assigns a capability to an object when a variable pointing to it is annotated. Any future objects that are stored in the local variable after the object that has been annotated must once again have an annotation present to be assigned a capability or they will be considered unsafe. Once an object has been assigned a capability, it retains that capability—or a stricter one—for its entire lifetime.

A secondary mechanism applies only to `@Immutable` capabilities and involves defining immutable classes globally. Classes listed in `resources/immutable-classes.txt` are automatically treated as immutable. This approach is suitable for Java classes that are intrinsically immutable, such as `String`, `Boolean`, `Integer`, and `Byte`. Note that all primitive types are treated as `@Immutable` by default.

Recognizing these classes as `@Immutable` is essential, as it enables them to be safely included within `@Local` or `@Isolated` objects without requiring explicit annotation. Without this class-wide designation, such intrinsically immutable types—despite their known immutability —would otherwise be treated as unsafe, potentially restricting their use in contexts where immutability is a requirement.

D. Object Initialisation Protocol

Constructors provide two unique challenges. The first is caused by a Java bytecode optimisation that allows object fields to be set before an object's constructor is called. This isn't allowed in Java source code [5] However, compilers can still generate such bytecode. At this early stage in the constructor, the object has not yet completed initialisation and does not fully extend `Object`. In the Java bytecode, this incomplete state is represented using the special `UninitializedThis` value instead of the standard `this` reference. As a result, it cannot yet be treated as a fully valid object, which complicates instrumentation and capability tracking during construction. To deal with this special case, JDala's instrumentation checks whether field access occurs in a constructor, and if so will temporary store values to be checked in local variables. As soon as the super-constructor is called, JDala will perform all of the validation checks at once for those values. This means that in some cases the line numbers in stack traces in JDala exceptions created when those checks fail might be off by a few lines.

The second challenge is related to immutable classes. Unlike annotated objects, which are registered as immutable only after their construction, globally defined immutable classes

[3] *Safe* objects are objects annotated using either `@Immutable`, `@Isolated` or `@Local`, all other objects are referred to as *unsafe* objects.

[4] The capabilities considered here form a hierarchy, see [14] for details.

[5] Except JEP447 (https://openjdk.org/jeps/447) which allows statements before `super(...)`, JEP447 is current at preview stage.

are considered immutable by default and are therefore subject to capability enforcement from the outset. This creates a complication: during construction, these classes must be allowed to modify their internal state, but doing so must not compromise the overall soundness of the system, particularly with respect to other objects accessed within the constructor. To address this, JDala introduces a targeted exception to its enforcement rules: it permits an object to modify its own fields within its own constructor, regardless of whether it is marked as `@Immutable`. This exemption allows immutable objects to be initialized correctly while preserving capability safety for interactions with other objects during construction.

E. Static Fields

Static fields are shared across all instances of a class and are therefore considered class-level, rather than object-level. Modifications to static fields are not captured by JDala's object instrumentation.

F. Memory Leak Prevention

A key implication of using global data structures for capability tracking is the potential for memory leaks. Because references are stored in maps accessed through static variables, those objects may become eligible for garbage collection. To mitigate this, weak references are employed within these collections, allowing unused objects to be reclaimed by the JVM's garbage collector when no strong references remain.

G. Reflection Support

Java reflection can be used to bypass conventional field access patterns, posing a challenge to capability enforcement. To address this, JDala instruments the `Field::get` and `Field::set` methods. This ensures that field modifications performed via reflection still trigger the appropriate capability checks.

There are other dynamic programming patterns that could be used to bypass the capability contracts. In particular, JDala does not currently restrict the use of `Unsafe` [35]. However, direct use of `Unsafe` by applications is discouraged and restricted in newer versions of Java.

H. Modelling the Transfer of `@Isolated` objects

Dala allows for isolated objects to be transferred between threads. Any isolated object that moves from one thread to another will lose its affiliation with the first thread. This principle can be recreated in Java by setting a reference to `null` after an object has moved. This could lead to a non-descriptive `NullPointerException` later in the code.

For `@Isolated` objects JDala allows multiple references to exist in one thread, with checks being performed that the object remains in its associated thread. `@Isolated` objects are initially associated with the thread they were created in. For transferring an `@Isolated` object to a new thread a dedicated *portal object* must be used. Portal object protocols have to be defined by specifying methods for objects to *enter* the portal (therefore being disconnected from the current thread) and methods to *exit* the portal (therefore becoming associated with the current thread). While it is possible to define portal objects by annotating them, with properties to define the respective enter and exit methods, we deemed this too complex. Most transfers follow patterns where portal objects are instances of dedicated *portal classes* like blocking queues. These classes are defined in the resource `portal-classes.json`, a classical example is `java.util.concurrent.BlockingQueue` with `put/offer/add` entry and `poll/take/remove` exit methods. Defining a particular type as a portal also applies to its subtypes.

If an isolated object enters a portal it goes into a *transfer state*. In this state the object cannot be accessed [6] by any thread. The object can then leave the portal via one of the predefined exit methods and at this point, it leaves the transfer state and becomes owned by the thread invoking the exit method.

I. Instrumentation Scope and Shading

Common challenges when building agents are self-instrumentation and instrumentation of internal utility classes, such as the data structures used to track capability-annotated objects. Instrumenting these classes directly could result in unintended recursive instrumentation cycles, potentially compromising runtime stability. To address this, JDala uses *shading* to incorporate private, namespaced copies of `org.json`, `org.plumelib.util`, and a concurrency map wrapper from `java.util.Collections`. These shaded versions are explicitly excluded from instrumentation. To maintain soundness, these shaded classes are to be used exclusively by JDala classes and should not be accessed by other parts of the application.

In addition, several `--add-opens` flags have been introduced to the project's Maven configuration. These flags explicitly grant reflective access to internal Java platform modules that are otherwise inaccessible under the Java Module System. This access is essential for JDala to perform bytecode instrumentation, intercept field and method accesses, and monitor runtime behaviour across a wide range of classes. Without these flags, the agent would encounter `IllegalAccessException` at runtime, or fail to apply the necessary instrumentation to enforce capability semantics reliably.

V. Usage and Evaluation

A. Building and Using JDala

JDala can be obtained from GitHub by cloning https://github.com/jensdietrich/jdala/.

JDala requires two rounds of compiling, the first one to build the agent, and the second to build the code that is used. The first build creates the agent but skips the tests:

```
mvn clean package -DskipTests
```

[6] Object fields can neither be read nor mutated.

The reason is that most tests need the agent to function. Tests can be executed in a second build that uses the Java agent created by the first build as value of the `-javaagent` JVM argument.

B. Evaluation

The tool is designed to be a Technology Readiness Level 4 [31] and has not been fully evaluated against real world data. Evaluation against relevant benchmarks, such as *Jacontebe* [28] is future work. This would require the manual annotation of benchmark code. There is however a comprehensive test suite included in the project.

VI. Conclusion

We have presented JDala, a proof-of-concept implementation of the Dala capability model, built atop the Java programming language. The implementation demonstrates that novel programming language concepts can be retrofitted into existing mainstream languages, thereby promoting their broader adoption. The JDala framework provides a platform for experimenting with capability-based security in the context of real-world applications.

References

[1] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: the absence of (inessential) difficulty. In *Onwards!*, 2012.

[2] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002.

[3] Elias Castegren and Tobias Wrigstad. Reference capabilities for concurrency control. In *ECOOP*, 2016.

[4] David Chisnall, Matthew Parkinson, and Sylvan Clebsch. Project Verona. `www.microsoft.com/en-us/research/project/-project-verona`, 2021.

[5] David Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, 1998.

[6] Sylvan Clebsch et al. Deny capabilities for safe, fast actors. In *AGERE*, pages 1–12, 2015.

[7] Michael J. Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Can advanced type systems be usable? an empirical study of ownership, assets, and typestate in Obsidian. *OOPSLA*, 2020.

[8] Russ Cox, Robert Griesemer, Rob Pike, Ian Lance Taylor, and Ken Thompson. The go programming language and environment. *Commun. ACM*, 65(5):70–78, April 2022.

[9] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in AmbientTalk. In *ECOOP*, pages 230–254, 2006.

[10] Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about object capabilities with logical relations and effect parametricity. In *EuroS&P*, 2016.

[11] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.

[12] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[13] Michael D Ernst and Mahmood Ali. Building and using pluggable type systems. In *FSE*, 2010.

[14] Kiko Fernandez-Reyes, Isaac Oscar Gariano, James Noble, Erin Greenwood-Thessman, Michael Homer, and Tobias Wrigstad. Dala: a simple capability-based dynamic language design for data race-freedom. In *Onward!*, 2021.

[15] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Dependeable Systems and Networks (DSN)*, pages 221–230, 2010.

[16] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *EuroSys*, page 328–343, 2017.

[17] Shin Hong and Moonzoo Kim. A survey of race bug detection techniques for multithreaded programmes. *Software Testing, Verification and Reliability*, 25(3):191–217, 2015.

[18] Vivian Hu. Rust breaks into TIOBE top 20 most popular programming languages. InfoQ, June 2020.

[19] DeLesley Hutchins, Aaron Ballman, and Dean Sutherland. C/C++ thread safety analysis. In *SCAM*.

[20] B. Jacobs, K. R. M. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Software Engineering and Formal Methods (SEFM)*, pages 137–146, 2005.

[21] James Noble, John Potter, and Jan Vitek. Flexible alias protection. In *ECOOP*, July 1998.

[22] C. B. Jones. Tentative steps toward a development method for interfering programs. *TOPLAS*, 1983.

[23] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. 2nd edition, 2018.

[24] Paul Krill. Microsoft forms Rust language team. InfoWorld, February 2021.

[25] Shriram Krishnamurthi and Tim Nelson. The human in formal methods. In *iFM*, 2019.

[26] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 2nd edition, December 1998.

[27] K Rustan M Leino and Peter Müller. Verification of concurrent programs with Chalice. In *ESOP*, 2009.

[28] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. Jacontebe: A benchmark suite of real-world java concurrency bugs (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 178–189. IEEE, 2015.

[29] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *CACM*, 58(2):44–46, 2015.

[30] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, 2008.

[31] John C Mankins et al. Technology readiness levels. *White Paper, April*, 6(1995):1995, 1995.

[32] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. Javacop: Declarative pluggable types for java. *TOPLAS*, 32(2):1–37, 2010.

[33] Stefan Marr. Why is concurrent programming hard? `www.stefan-marr.de/2014/07/-why-is-concurrent-programming-hard`, July 2014.

[34] Stefan Marr and Hanspeter Mössenböck. Optimizing communicating event-loop languages with Truffle. In *AGERE*, October 2015.

[35] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The java unsafe api in the wild. *ACM Sigplan Notices*, 50(10):695–710, 2015.

[36] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, Maryland, 2006.

[37] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, volume 8, 2008.

[38] James Noble. Dafny vs. dala: Experience with mechanising language design. In *Workshop on Formal Techniques for Java-like Programs*, 2024.

[39] Jim Shore. Fail fast [software debugging]. *IEEE Software*, 21(5):21–25, 2004.

[40] Fridtjof Stoldt, Brandt Bucher, Sylvan Clebsch, Matthew A. Johnson, Matthew J. Parkinson, Guido van Rossum, Eric Snow, and Tobias Wrigstad. Dynamic region ownership for concurrency safety. In *PLDI*, 2025.

[41] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In *ICSE*, 2020.

[42] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.

[43] Stephan van Staden. On rely-guarantee reasoning. In *MPC*, pages 30–49, 2015.