Coping with shared mutable state in a typestate-oriented concurrent language

ABSTRACT

In this PhD thesis plan, we propose to design a type system for a core **object** and **typestate-oriented** language to reason on **shared mutable state** while ensuring *memory* and *thread-safety*, *protocol compliance* (that method calls are performed in order) and *completion* (if the program terminates, all protocols are finished and resources are freed). We plan to address this by implementing a **type system framework** parametric over **separation algebras** allowing one to distinguish between *thread-local* and *thread-shared* resources, and between *affine* and *linear* ones.

1 CONTEXT

Users expect applications to be well-behaved and efficient. However, all sorts of bugs appear in software. Issues compromising memory-safety, like null dereferencing [17], dangling pointers, or memory leaks, lead to crashes. Since concurrency is intrinsic to modern software and hardware, issues affecting thread-safety, like data-races or race-conditions [36], lead to subtle bugs [29]. Protocol-related bugs [4, 38], like executing methods out of order, or forgetting to call a finalizer method, result in programs that fail to follow the intended behaviour, leading to business logic issues or even security ones [11–13, 28]. For efficiency, modern software relies heavily on imperative and concurrent language features, exploiting shared mutable state. However, reasoning about these programs is very difficult [26, 29]. Thus, software verification is crucial to ensure programs are correct with respect to their intended behaviour [18].

Most static verification techniques are based either on type systems or deductive logics. *Type systems* are widely used in industrial languages to avoid data-errors. However, most do not prevent critical bugs, like null dereferencing, such as C and Java. More modern languages have adopted richer type systems that avoid more errors: Kotlin distinguishes nullable from non-nullable types; Rust's ownership model guarantees memory and thread-safety. Nonetheless, mainstream languages still do not enforce relevant properties, such as guaranteeing that object protocols are followed and completed. Behavioural type disciplines, like typestates [14, 16, 40, 41] and session types [19, 20, 42], have been proposed and thoroughly studied to fill this gap, but are not used in the industry. *Deductive reasoning* is not integrated in compilers; nonetheless, it is used to verify properties that cannot be expressed in common type systems, like functional correctness. For instance, the VeriFast tool verifies C and Java code given separation logic specifications [22].

2 PROBLEMS

In the literature, there are many approaches to static program verification. However, each solution is usually focused on a particular issue and is limited to a fixed set of features. So, there is no *unified system* which combines the qualities of several approaches while accounting for modern features. For example, dynamic thread creation is usually not accounted for in high-level languages, while parallel composition, which is not used in modern languages, is subject of more research, as Dodds et al. [15] point out. Moreover, there are useful disciplines in the literature that are not then implemented in mainstream programming languages, such as behavioural types [1].

There is also a tension between expressiveness and proof ease. *Type systems* have a lower annotation effort than deductive logics but are usually less expressive. One may need unnecessary workarounds to show that the code is correct, like rewriting it in an unnatural way, using defensive programming, or using locks in sequential code. In the worst case, the code might not be accepted. *Deductive logics* are more expressive but may require challenging proofs or cumbersome encodings [33]: the developer wastes time, and the correctness proof intuition gets lost in the encoding.

Thus, programming languages should ideally provide: (1) a *unified framework* to check safety of code supporting several modern features – connecting what exists in the literature with what is used in the industry; (2) a *balance* between expressiveness and proof ease, exploiting low specification efforts and good abstractions – improving developer experience. We consider that a *decidable type system* can provide the right balance since the programmer would not be required to provide proofs. However, a *unified framework* to deal with **protocols** and **shared mutable state** in type systems is still lacking and there are problems that still need to be addressed. In particular, we identify three concrete unsolved issues:

1. Sharing patterns are severely limited: either sharing is forbidden (by enforcing linearity[1]), or limited to a fixed set of capabilities, or a certain ownership discipline, preventing circular data structures from being implemented;
2. Thread-local data and thread-shared data are not differentiated, forcing the use of locks even in sequential code;
3. Protocol completion is supported only in linear settings. In the presence of sharing, affinity[2] has been preferred, leading to memory leaks or uncompleted protocols.

To highlight these points, please consider, for example purposes, the JavaScript asynchronous code [27, 34, 39] in List. 1 featuring a producer and consumer. Both share a queue (implemented as an linked list) containing stateful objects with protocol (line 1). The producer performs a task and adds the result to the queue (lines 4-5). The consumer keeps taking values while there are objects to consume or while a producer is requesting the queue for future additions (lines 10-11). Due to the interleaving of actions and the sharing pattern exhibited, which relies on a complex cooperation between the producer and consumer (i.e. the producer adds items expecting the consumer to receive them all and complete their protocols), verifying that the protocols are respected and completed is not trivial. If the producer requested the queue in line 5 instead of line 16, it would be possible for the consumer to execute first, observe that the queue was empty and unused, and terminate immediately. After the producer finished, the whole program would terminate with the queue having unconsumed objects.

[1] Which only allows one reference to data.

[2] Which allows data to not be used (i.e. dropped).

Listing 1: Asynchronous producer and consumer

```
const queue = new Queue();

async function producer() {
  const data = await otherTask1();
  queue.add(data);
  queue.unuse();
}

async function consumer() {
  while (queue.inUse()) {
    const data = await queue.take();
    await otherTask2(data);
  }
}
// Claim use before starting the consumer
queue.use();
await Promise.all([producer(), consumer()]);
```

Issue 1 prevents the verification of the queue and controlling the states of the objects [33]. Some approaches might be able to tackle this, but at the cost of either not having a fine-grained resource control, necessary to ensure protocol completion (Issue 3), or by requiring the use of deductive reasoning. Moreover, as explained before, the sharing pattern relies on a complex cooperation between the producer and consumer which is difficult to model while ensuring protocol completion. How to tame mutable shared state has been usually studied either in sequential settings or multi-threaded programs. For this reason, unless the program is fully sequential, it is usually assumed that data may be shared between threads, forcing one to use some form of synchronization to access it (Issue 2). Finally, as far as we can tell, principled techniques to better handle mutable shared state in single-threaded asynchronous settings have not been proposed (where the interleaving of asynchronous calls leads to concurrency without parallelism[3] - thanks to the "run to completion" scheduling of event loops [10, 31, 34, 39, 43]).

The borrowing rules of Rust [44] do not support this scenario. One would likely need to use locks to control the access to the queue and reference counting to know when to drop the queue, moving the verification to run time. Even with a library[4], the types are fixed so there is no notion of protocol. CLASS [37] does not support fine-grained resource control or linear state in cells. Access permissions of Plural [5] are not expressive enough to model this kind of cooperation. Rely-guarantee protocols [30] could be used to model the example, but locks are required in concurrent settings and protocol completion is not guaranteed. The aforementioned issues have been addressed in some settings, but still only partially:

(1) With the proliferation of solutions to tackle issues of expressiveness, each one with their own advantages and disadvantages, a unified framework is desired. Iris, a framework for higher-order concurrent separation logic [24], allows users to implement their own logical (ghost) resources (as partial commutative monoids), and has been successfully used to derive and implement many different formal systems. Although Iris is an unifying and expressive framework, deductive reasoning and expertise are required.

(2) There is some work on capabilities which distinguish between thread-local from thread-shared data [8, 9, 45], but the set of available capabilities is fixed and there are limitations to how data may be transferred between threads. In Iris, it is possible to encode "thread-local invariants" which can be opened non-atomically [21], but doing the encoding is non-trivial and requires expert users.

(3) Recent logics have preferred to use affine resources, such as Iris, which do not allow the precise tracking of resources. However, there is now some work extending Iris with linear resources [23], or even both kinds [7, 25]. Unfortunately, this is only available in the deductive logics realm.

3 RESEARCH STATEMENT

Our main objective for the PhD thesis is to:

> *Design a typed core OO language supporting shared mutable state and objects protocols, with memory and thread-safety, protocol compliance and completion.*

The core language should provide modern features, support **protocols**, and reason about **shared mutable state**. Safe programs should be *memory-safe* (i.e. no null dereferencing, dangling pointers or memory leaks) and *thread-safe* (i.e. no low-level data-races). Moreover, safe programs should respect all objects protocols (*protocol compliance*) and ensure that upon termination all protocols are finished (*protocol completion*). The former is crucial to ensure that methods are executed in the right sequence. The latter is critical to guarantee that necessary method calls are not forgotten and resources are freed. To fulfil this goal, we plan the following.

Formalise the language semantics supporting modern features, like aliasing, mutable state, locks, dynamic thread creation (with `fork` and `join`), and asynchronous code (enabled by each thread having an event loop, inspired by AmbientTalk [10] and JavaScript [34, 39]), in the Coq proof assistant [3].

Develop a type system framework *parametric* over *separation algebras* [6], allowing more expressive ways to reason about shared data. Taking inspiration from Iris [24], we want to ease the creation of new type systems, without requiring one to repeat soundness proofs: one just needs to instantiate the framework with the right sharing capabilities. We plan to mechanise the solution in the Coq proof assistant [3], using computer-aided proofs to establish the soundness of the approach. With this *type system based approach*, we believe we provide a much needed balance between expressiveness and ease of use while providing an *unifying framework* from which more works can be developed, to avoid adding to "the next 700 type systems" [35]. The *technical novelties* would include the ability to distinguish between thread-local and thread-shared resources, and between affine and linear ones. Moreover, we will develop a *decidable algorithm* from the rules of the resulting type system.

Evaluate the approach by applying the principles in mainstream languages, like TypeScript or Java. JaTyC [2, 32], a Java typestate-checking tool, has been our test ambient. It statically ensures memory-safety, protocol compliance and completion. However, objects must be used linearly. To support flexible sharing, we also plan to develop an integration of typestates with particular *separation algebras*, such as access permissions [5] and rely-guarantee protocols [30], thus going beyond the state of the art.

[3] As Cutsem et al. [10] point out, "the use of event loops avoids low-level data races that are inherent in the shared-memory multithreading paradigm".

[4] https://doc.rust-lang.org/std/sync/mpsc/fn.channel.html

REFERENCES

[1] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Found. Trends Program. Lang.* 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031

[2] Lorenzo Bacchiani, Mario Bravetti, Marco Giunti, João Mota, and António Ravara. 2022. A Java typestate checker supporting inheritance. *Sci. Comput. Program.* 221 (2022), 102844. https://doi.org/10.1016/J.SCICO.2022.102844

[3] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. 1997. *The Coq proof assistant reference manual: Version 6.1.*

[4] Nels E Beckman, Duri Kim, and Jonathan Aldrich. 2011. An Empirical Study of Object Protocols in the Wild. In *Proc. of European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science, Vol. 6813)*. Springer, United Kingdom, 2–26. https://doi.org/10.1007/978-3-642-22655-7_2

[5] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 301–320. https://doi.org/10.1145/1297027.1297050

[6] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society, 366–378. https://doi.org/10.1109/LICS.2007.30

[7] Qinxiang Cao, Santiago Cuellar, and Andrew W. Appel. 2017. Bringing Order to the Separation Logic Jungle. In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10695)*, Bor-Yuh Evan Chang (Ed.). Springer, 190–211. https://doi.org/10.1007/978-3-319-71237-6_10

[8] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:26. https://doi.org/10.4230/LIPICS.ECOOP.2016.5

[9] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos A. Varela (Eds.). ACM, 1–12. https://doi.org/10.1145/2824815.2824816

[10] Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. 2014. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Comput. Lang. Syst. Struct.* 40, 3-4 (2014), 112–136. https://doi.org/10.1016/J.CL.2014.05.002

[11] CWE - Common Weakness Enumeration. 2023. CWE-306: Missing Authentication for Critical Function. https://cwe.mitre.org/data/definitions/306.html Accessed: 2024-01-22.

[12] CWE - Common Weakness Enumeration. 2023. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html Accessed: 2024-01-22.

[13] CWE - Common Weakness Enumeration. 2023. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html Accessed: 2024-01-22.

[14] Robert DeLine and Manuel Fähndrich. 2004. Typestates for Objects. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3086)*, Martin Odersky (Ed.). Springer, 465–490. https://doi.org/10.1007/978-3-540-24851-4_21

[15] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 363–377. https://doi.org/10.1007/978-3-642-00590-9_26

[16] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 36, 4 (2014), 12:1–12:44. https://doi.org/10.1145/2629609

[17] Tony Hoare. 2009. Null References: The Billion Dollar Mistake. https://tinyurl.com/eyipowm4 Presentation at QCon London.

[18] Tony Hoare, Jayadev Misra, Gary T. Leavens, and Natarajan Shankar. 2021. The Verified Software Initiative: A Manifesto. In *Theories of Programming: The Life and Works of Tony Hoare*, Cliff B. Jones and Jayadev Misra (Eds.). ACM Books, Vol. 39. ACM / Morgan & Claypool, 81–92. https://doi.org/10.1145/3477355.3477361

[19] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 715)*, Eike Best (Ed.). Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

[20] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1381)*, Chris Hankin (Ed.). Springer, 122–138. https://doi.org/10.1007/BFB0053567

[21] Iris Project. 2023. The Iris 4.1 Reference. https://plv.mpi-sws.org/iris/appendix-4.1.pdf Accessed: 2024-01-15.

[22] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

[23] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2024. Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 1385–1417.

[24] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[25] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

[26] Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2018. A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs. In *Programming with Actors - State-of-the-Art and Research Perspectives*, Alessandro Ricci and Philipp Haller (Eds.). Lecture Notes in Computer Science, Vol. 10789. Springer, 155–185. https://doi.org/10.1007/978-3-030-00302-9_6

[27] Matthew C. Loring, Mark Marron, and Daan Leijen. 2017. Semantics of asynchronous JavaScript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages, Vancouver, BC, Canada, October 23 - 27, 2017*, Davide Ancona (Ed.). ACM, 51–62. https://doi.org/10.1145/3133841.3133846

[28] Gavin Lowe. 1996. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1055)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 147–166. https://doi.org/10.1007/3-540-61042-1_43

[29] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. 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, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, Susan J. Eggers and James R. Larus (Eds.). ACM, 329–339. https://doi.org/10.1145/1346281.1346323

[30] Filipe Militão, Jonathan Aldrich, and Luís Caires. 2014. Rely-Guarantee Protocols. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 334–359. https://doi.org/10.1007/978-3-662-44202-9_14

[31] Mark S. Miller, Eric Dean Tribble, and Jonathan S. Shapiro. 2005. Concurrency Among Strangers. In *TGC (Lecture Notes in Computer Science, Vol. 3705)*. Springer, 195–229.

[32] João Mota, Marco Giunti, and António Ravara. 2021. Java Typestate Checker. In *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12717)*, Ferruccio Damiani and Ornela Dardha (Eds.). Springer, 121–133. https://doi.org/10.1007/978-3-030-78142-2_8

[33] João Mota, Marco Giunti, and António Ravara. 2023. On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage (Experience Paper). In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 40:1–40:29. https://doi.org/10.4230/LIPICS.ECOOP.2023.40

[34] Mozilla. 2024. The event loop - JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop Accessed: 2024-02-29.

[35] Matthew J. Parkinson. 2010. The Next 700 Separation Logics - (Invited Paper). In *Verified Software: Theories, Tools, Experiments, Third International Conference,*

VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6217), Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani (Eds.). Springer, 169–182. https://doi.org/10.1007/978-3-642-15057-9_12

[36] Kevin Poulsen. 2004. Tracking the blackout bug. Securityfocus.com. https://web.archive.org/web/20110610163731/http://www.securityfocus.com/news/8412 Retrieved June 11, 2010.

[37] Pedro Rocha and Luís Caires. 2023. Safe Session-Based Concurrency with Shared Linear State. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, France, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13990)*, Thomas Wies (Ed.). Springer, 421–450. https://doi.org/10.1007/978-3-031-30044-8_16

[38] Syeda Khairunnesa Samantha, Shibbir Ahmed, Sayem Mohammad Imtiaz, Hridesh Rajan, and Gary T. Leavens. 2023. What kinds of contracts do ML APIs need? *Empir. Softw. Eng.* 28, 6 (2023), 142. https://doi.org/10.1007/S10664-023-10320-Z

[39] Thodoris Sotiropoulos and Benjamin Livshits. 2019. Static Analysis for Asynchronous JavaScript Programs. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs, Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:30. https://doi.org/10.4230/LIPICS.ECOOP.2019.8

[40] Robert E. Strom. 1983. Mechanisms for Compile-Time Enforcement of Security. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*, John R. Wright, Larry Landweber, Alan J. Demers, and Tim Teitelbaum (Eds.). ACM Press, 276–284. https://doi.org/10.1145/567067.567093

[41] Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

[42] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-based Language and its Typing System. In *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings (Lecture Notes in Computer Science, Vol. 817)*, Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis (Eds.). Springer, 398–413. https://doi.org/10.1007/3-540-58184-7_118

[43] Andrew S. Tanenbaum. 2014. *Modern operating systems, 4rd Edition.* Pearson Prentice-Hall.

[44] The Rust Team. 2017. Rust Programming Language. https://www.rust-lang.org/ Accessed: 2023-12-15.

[45] Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. 2009. Loci: Simple Thread-Locality for Java. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5653)*, Sophia Drossopoulou (Ed.). Springer, 445–469. https://doi.org/10.1007/978-3-642-03013-0_21