—Position Paper—

Towards Meta-Level Engineering and Tooling for Complex Concurrent Systems

Stefan Marr* Elisa Gonzalez Boix† Hanspeter Mössenböck*

*Johannes Kepler University Linz, Austria
stefan.marr@jku.at, hanspeter.moessenboeck@jku.at
†Vrije Universiteit Brussel, Belgium,
egonzale@vub.ac.be

Abstract

With the widespread use of multicore processors, software becomes more and more diverse in its use of parallel computing resources. To address all application requirements, each with the appropriate abstraction, developers mix and match various concurrency abstractions made available to them via libraries and frameworks. Unfortunately, today's tools such as debuggers and profilers do not support the diversity of these abstractions. Instead of enabling developers to reason about the high-level programming concepts, they used to express their programs, the tools work only on the library's implementation level. While this is a common problem also for other libraries and frameworks, the complexity of concurrency exacerbates the issue further, and reasoning on the higher levels of the concurrency abstractions is essential to manage the associated complexity.

In this position paper, we identify open research issues and propose to build tools based on a common meta-level interface to enable developers to reasons about their programs based on the high-level concepts they used to implement them.

1 Introduction and Background

With the start of the multicore era, concurrent programming became more and more important. Unfortunately, it also complicates the software development significantly. Common concurrency problems include correctness issues such as data races and deadlocks as well as performance issues caused by resource contention and sequential bottlenecks. To avoid these problems, a wide variety of different concurrency models have been proposed.

Van Roy and Haridi [2004] categorize the main models into declarative, message-passing, and shared-state concurrency. Shared-state concurrency includes, e. g., software transactional memory (STM) [Shavit and Touitou, 1995] and thread-based models. Message-passing concurrency includes, e. g., the actor model [Agha, 1986] and communicating sequential processes [Hoare, 1978]. In the category of declarative concurrency fall approaches such as various kinds of data flow programming [Johnston et al., 2004] and more recent reactive programming approaches [Bainomugisha et al., 2013].

Each of these concurrency models solves its own set of problems. Consequently, applications started to combine a variety of concurrency models to address different functional and performance requirements. A recent study

Submission to: 9. Arbeitstagung Programmiersprachen, Vienna, Austria, 25 February 2016

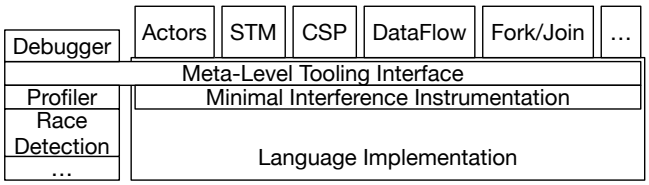

Figure 1: A meta-level interface to support tools for a wide range of concurrency abstractions.

of Scala programs [Tasharofi et al., 2013] investigated why developers use thread-based abstractions in actor-based applications. While the actor model avoids race conditions by design, because actors cannot share mutable state, this property can severely affect performance. Thus, developers combine it with threads. In addition, some problems are expressed more naturally by defining computations on shared memory. Unfortunately, combining concurrency models can lead to unexpected bugs, e. g., race conditions or deadlocks [Swalens et al., 2014].

While the use of multiple concurrency models in the same application has been recognized as an issue [Haller, 2015, Swalens et al., 2014, Tasharofi et al., 2013], the question of how to engineer and debug such systems has not been approached. Yet software tools are an integral part of application development since they help developers to cope with the complexity of software. On the one hand, tools such as inspectors, debuggers, and profilers need to be able to expose the high-level concurrency concepts such as transactions or asynchronous messages to the developers. On the other hand, tools need to overcome the fundamental performance overhead of collecting the runtime information on data access patterns, data races, conflicting updates in transactions, or actors that are sequential bottlenecks. To aid the development of complex concurrent applications, high-level debuggers are sorely needed to help programmers to find errors as well as to achieve a better understanding of the dynamic application behavior.

2 A Meta-Level Interface to Enable Tooling of Complex Concurrent Systems

To bring tooling to the wide range of concurrency models, we envision a meta-level interface that allows tools to connect to the language implementation and observe program execution as well as interact with it by using a common set of basic concepts that underlie a wide range of concurrency abstractions. This idea is reminiscent to classic metaobject protocols (MOP) [Kiczales et al., 1991], which reify the concepts of a language and its implementation to enable tooling as well as dynamic adaptation from within a program. This idea can also be seen in widely used VM tooling interfaces [Würthinger et al., 2010].

Figure 1 visualizes the relation of the envisioned meta-level interface with respect to concurrency abstractions, tools, and instrumentation at the level of language implementation. The key for making the interface independent of concrete concurrency abstractions is to identify the fundamental building blocks for each abstraction. For instance for message-passing approaches, the exchange of messages and the involved mailboxes or channels are essential, because they allow developers to reason about the information flow between actors or processes. Shared memory models on the other hand have other basic notions. In an STM, it is relevant to know in which order transactions have been executed, which ones caused transaction aborts or retries. To understand applications with locks, one needs however information on which resources are protected by locks, and how and when the locks are acquired. Declarative concurrency models are again different, in that they typically express data dependencies explicitly.

3 Open Research Issues

For such a meta-level interface, we need to distill the set of basic building blocks instead of merely enumerating all high-level concepts. Thus, one of the challenges is to identify the common core concepts shared between concurrency abstractions so that a wide range of concepts can be represented. However, this is not the only challenge. The next section details the problems that need to be tackled to realize the vision of a common meta-level interface for debugging.

To build high-level debuggers and other tools for the development of complex concurrent systems, we face issues in three areas: how to debug programs that employ multiple concurrency models, how to provide instrumentation for such tooling that minimizes interference with program execution, and how to decouple these mechanisms from concrete concurrency abstractions. We detail these problem briefly in the remainder of this section.

3.1 Debuggers for Programs Employing Multiple Concurrency Models

Prior research has studied debugging support for many of the individual concurrency models. However, debuggers for programs that employ multiple concurrency models are unexplored. Generally, McDowell and Helmbold [1989] distinguish classic breakpoint-based debuggers and event-based ones. Event-based debuggers typically record the history of events in an application and might provide analyses on top of these events. Depending on the concurrency model, an event may be an API call, a memory load/store, or a message send/receive.

As an approach that is especially relevant in the field of concurrency, research on breakpoint-based debuggers led to time-traveling debuggers [Barr and Marron, 2014, Pluquet et al., 2009, Pothier et al., 2007], which preserve the dynamic program history to enable forward and backward step-by-step execution and state inspection. Focusing on specific models, breakpoint-based debuggers have been investigated for thread-based concurrency models, STM [Zyulkyarov et al., 2010], and event loops [Gonzalez Boix et al., 2011].

However, it remains open how to build a uniform debugger that supports a wide variety of concurrency models. We see this as an open question both from the user perspective and the implementation perspective. Considering typical scenarios, such a debugger needs to be able to help developers to understand their programs from different perspectives or levels of granularity. For instance in an actor program, one might want to continue execution until a message is processed completely, or one might want to see which messages caused which effects to see races or bottlenecks. In a transaction, one might want to complete a whole atomic block, or in a lock-based part of the program continue execution until a lock is taken or released. All these are different concepts that have to be carefully represented to be understandable for the user, but must also be realized in a way generic that is enough so that the new en vogue concurrency concept of the day can be supported.

3.2 Low-Interference Program Instrumentation

For debugging, it is also still an open question how to record concurrent executions, while minimizing the necessary data and instrumentation. Instrumentation of concurrent programs is generally problematic, because it can interfere with the execution so that data races might never appear with the instrumentation. Thus, it is essential to minimize general interference. Part of that is to minimize the amount of recorded events [Barr and Marron, 2014, Hofer et al., 2016, Lengauer et al., 2015, Pothier et al., 2007]. This could for instance go in the direction of techniques for debugging message-passing systems. Honda and Yonezawa [1988] propose to raise the abstraction level to debug systems on the level of object groups. This approach structures the event history in terms of the tasks performed collectively by a group of objects, which is one relevant approach to present the high-level concepts to developers.

The main goal is, however, to identify techniques that are applicable to a wide range of concurrency models to be able to represent high-level concepts to developers. Thus, relevant techniques include recording for replay [Liu et al., 2015, Prähofer et al., 2011], concurrency bug reproduction [Huang and Zhang, 2012], dataflow [Tripp et al., 2011] and data dependency discovery [Bond et al., 2013], as well as recording of performance-related information [David et al., 2014, Hofer et al., 2015a]. So far, however, the known precise approaches cause interferences in the execution or do not scale to complex systems. Imprecise, i. e., probabilistic approaches are potentially a solution. Existing work investigates the use of hardware support [Hofer et al., 2015b, Inoue and Nakatani, 2009], and discusses how to derive high-level information [Jin et al., 2010, Mytkowicz et al., 2009], but has not yet been investigated in the context of multiple high-level concurrency models.

3.3 Decoupling Infrastructure from Concrete Concurrency Models

As mentioned in section 2, our goal is to define a meta-level interface that can be used to observe and to interact with the execution of complex concurrent systems that use a wide range of concurrency abstractions. We envision the use of a meta-level interface based on the insight that tools always have a tight connection to meta-level programming. Thus, it is natural that programs are either manipulated (statically) or executed and monitored (dynamically) by a tool, which can also be the program itself. The code of the tool interacts with the code of the program by means of such a meta-level interface, which is tightly bound to the design of the programming language. Examples are VM tooling interfaces [Würthinger et al., 2010] and metaobject protocols (MOP) [Kiczales et al., 1991]. MOPs have been studied in the field of distributed computing [Cazzola, 2003, Garbinato et al., 1994, McAffer, 1995] as well as to realize individual concurrency models on the same VM [Marr and D'Hondt, 2012]. However, it is still open which basic concepts such an interface needs to make it possible to build a wide range of concurrency abstractions on top of it and to provide the necessary information about

interactions of concurrency abstractions for the purpose of building tools such as debuggers. Thus, we need to find ways to decouple the meta-level interface from concrete concurrency models, and to abstract from them.

4 Conclusion

For the development of complex concurrent systems, we need better tools! Currently, debugging and tooling is only available on the implementation level and does not capture high-level concepts. With the complexity of concurrent software especially when it mixes and matches different abstractions, it is essential to preserve these abstractions also during debugging in order to understand a program's behavior.

We see a meta-level interface that abstracts from concrete concurrency models as a way forward. However, there are many open research questions: starting from how to build debuggers for different interacting concurrency abstractions, over the question of how to avoid problematic interference with program execution, as well as how to abstract from concrete concurrency models for such an interface.

Recently, we started to investigate these issues in the context of the Truffle and Graal language implementation platform [Würthinger et al., 2013] of Oracle Labs. We intent to focus on this research, but with the large number of open questions, we welcome collaborations and exchange of ideas that can lead to solving today's issues of building complex concurrent applications.

Acknowledgements

This research is funded by a collaboration grant of the Austrian Science Fund (FWF) with the project I2491-N31 and the Research Foundation Flanders (FWO Belgium).

References

G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

E. Bainomugisha, A. Lombide Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, Aug. 2013. doi: 10.1145/2501654.2501666.

E. T. Barr and M. Marron. Tardis: Affordable time-travel debugging in managed runtimes. In *Proc. of OOPSLA*, pages 67–82. ACM, 2014. doi: 10.1145/2660193.2660209.

M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang. Octet: Capturing and controlling cross-thread dependences efficiently. In *Proc. of OOPSLA*, pages 693–712. ACM, 2013. doi: 10.1145/2509136.2509519.

W. Cazzola. Remote method invocation as a first-class citizen. *Distributed Computing*, 16(4):287–306, 2003. ISSN 0178-2770. doi: 10.1007/s00446-003-0094-8.

F. David, G. Thomas, J. Lawall, and G. Muller. Continuously measuring critical section pressure with the free-lunch profiler. In *Proc. of OOPSLA*, pages 291–307. ACM, 2014. doi: 10.1145/2660193.2660210.

B. Garbinato, R. Guerraoui, and K. Mazouni. Distributed programming in garf. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Programming*, volume 791 of *LNCS*, pages 225–239. Springer Berlin Heidelberg, 1994. doi: 10.1007/BFb0017543.

E. Gonzalez Boix, C. Noguera, T. Van Cutsem, W. De Meuter, and T. D'Hondt. Reme-d: A reflective epidemic message-oriented debugger for ambient-oriented applications. In *Proc. of SAC*, pages 1275–1281. ACM, 2011. doi: 10.1145/1982185.1982463.

P. Haller. High-Level Concurrency Libraries: Challenges for Tool Support, October 2015. Keynote of Eclipse Technology eXchange Workshop, collocated with SPLASH '15. `http://2015.splashcon.org/event/etx2015-keynote-by-philipp-haller`.

C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

P. Hofer, D. Gnedt, and H. Mössenböck. Efficient Dynamic Analysis of the Synchronization Performance of Java Applications. In *Proc. of the WODA Workshop*, pages 14–18. ACM, 2015a. doi: 10.1145/2823363.2823367.

P. Hofer, F. Hörschläger, and H. Mössenböck. Sampling-based Steal Time Accounting Under Hardware Virtualization. In *Proc. of ICPE*, pages 87–90. ACM, 2015b. doi: 10.1145/2668930.2695524.

P. Hofer, D. Gnedt, A. Schörgenhumer, and H. Mössenböck. Efficient Tracing and Versatile Analysis of Lock Contention in Java Applications on the Virtual Machine Level. In *Proc. of the ACM/SPEC Conf. on Performance Engineering (ICPE'16)*, March 2016.

Y. Honda and A. Yonezawa. Debugging concurrent systems based on object groups. In S. Gjessing and K. Nygaard, editors, *Proc. of ECOOP*, volume 322 of *LNCS*, pages 267–282. Springer Berlin Heidelberg, 1988. doi: 10.1007/3-540-45910-3_16.

J. Huang and C. Zhang. Lean: Simplifying concurrency bug reproduction via replay-supported execution reduction. In *Proc. of OOPSLA*, volume 47, pages 451–466. ACM, Oct. 2012. doi: 10.1145/2398857.2384649.

H. Inoue and T. Nakatani. How a java vm can get more from a hardware performance monitor. In *Proc. of OOPSLA*, pages 137–154. ACM, 2009. doi: 10.1145/1640089.1640100.

G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proc. of OOPSLA*, pages 241–255. ACM, 2010. doi: 10.1145/1869459.1869481.

W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, Mar. 2004. doi: 10.1145/1013208.1013209.

G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, MA, USA, 1991. ISBN 9780262610742.

P. Lengauer, V. Bitto, and H. Mössenböck. Accurate and efficient object tracing for java applications. In *Proc. of ICPE*, pages 51–62. ACM, 2015. doi: 10.1145/2668930.2688037.

P. Liu, X. Zhang, O. Tripp, and Y. Zheng. Light: Replay via tightly bounded recording. In *Proc. of PLDI*, pages 55–64. ACM, 2015. doi: 10.1145/2737924.2738001.

S. Marr and T. D'Hondt. Identifying a unifying mechanism for the implementation of concurrency abstractions on multi-language virtual machines. In *Proc. of TOOLS*, volume 7304 of *LNCS*, pages 171–186. Springer, May 2012. doi: 10.1007/978-3-642-30561-0_13.

J. McAffer. Meta-level programming with coda. In M. Tokoro and R. Pareschi, editors, *Proc. of ECOOP'95*, volume 952 of *LNCS*, pages 190–214. Springer Berlin Heidelberg, 1995. doi: 10.1007/3-540-49538-X_10.

C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, Dec. 1989. doi: 10.1145/76894.76897.

T. Mytkowicz, D. Coughlin, and A. Diwan. Inferred call path profiling. In *Proc. of OOPSLA*, pages 175–190. ACM, 2009. doi: 10.1145/1640089.1640102.

F. Pluquet, S. Langerman, and R. Wuyts. Executing code in the past: Efficient in-memory object graph versioning. In *Proc. of OOPSLA*, pages 391–408. ACM, 2009. doi: 10.1145/1640089.1640118.

G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proc. of OOPSLA*, pages 535–552. ACM, 2007. doi: 10.1145/1297027.1297067.

H. Prähofer, R. Schatz, C. Wirth, and H. Mössenböck. A comprehensive solution for deterministic replay debugging of softplc applications. *Industrial Informatics, IEEE Transactions on*, 7(4):641–651, November 2011. ISSN 1551-3203. doi: 10.1109/TII.2011.2166768.

N. Shavit and D. Touitou. Software transactional memory. In *Proc. of PODC '95*, pages 204–213. ACM, 1995. doi: 10.1145/224964.224987.

J. Swalens, S. Marr, J. De Koster, and T. Van Cutsem. Towards composable concurrency abstractions. In *Proc. of the PLACES Workshop*, volume 155, pages 54–60, April 2014. doi: 10.4204/EPTCS.155.8.

S. Tasharofi, P. Dinges, and R. E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *LNCS*, pages 302–326. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-39038-8_13.

O. Tripp, G. Yorsh, J. Field, and M. Sagiv. Hawkeye: Effective discovery of dataflow impediments to parallelization. In *Proc. of OOPSLA*, pages 207–224. ACM, 2011. doi: 10.1145/2048066.2048085.

P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, March 2004. ISBN 0262220695.

T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Proc. of Onward!*, pages 187–204. ACM, 2013. doi: 10.1145/2509578.2509581.

T. Würthinger, M. L. Van De Vanter, and D. Simon. Multi-level virtual machine debugging using the java platform debugger architecture. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 401–412. Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-11486-1_34.

F. Zyulkyarov, T. Harris, O. S. Unsal, A. Cristal, and M. Valero. Debugging programs that use atomic blocks and transactional memory. In *Proc. of PPoPP*, pages 57–66. ACM, 2010. doi: 10.1145/1693453.1693463.