Verifying Lock-Free Data Structures and Algorithms

Master's Thesis Project Description
Felix A. Wolf

Supervisors: Prof. Dr. Peter Müller, Dr. Malte Schwerhoff
Department of Computer Science, ETH Zürich

Start: 21st February 2018
End: 22nd August 2018

Context

Verifying sequential code in general requires reasoning about unbounded heap structures; each piece of code can potentially manipulate arbitrary parts of the heap, hence, to enable reasoning, a verification logic is required which is expressive enough to specify the heap effects of the code.

Separation logic allows us to treat the heap as a memory resource that can be subdivided into disjoint parts. The behavior of a piece of code, for example a method, can then be specified by additionally stating the manipulated heap resources. We refer to this concept as *resource disjointness*; the underlying semantics are that the ownership of the stated resources are transferred from the caller to the callee at invocation time and are returned when the call exits. Since code can only manipulate memory that it has ownership of and the ownership of resources itself is disjoint, local reasoning is enabled. Powerful tools have been built to automate this verification process [8].

Next is the verification of *concurrent* code. Because locks enforce mutual exclusivity, verification logics that incorporate resource disjointess are still sufficient [5] [7], if every shared memory access is protected by a lock. However, many high-performance applications eschew the potential runtime overhead of locks and instead rely on *lock-free data structures and algorithms*, implemented using atomic hardware instructions such as compare-and-swap. These more flexible synchronization techniques make specification, and thus verification, more complicated, because, in general, the heap resources manipulated by some code can be modified by another thread at any moment. A simple example is the increment method of a concurrent lock-free counter: the behavior of the increment method cannot be specified by the concept of resource disjointness alone, because the underlying semantics of resource disjointness, i.e. assigning the ownership of the counter heap resource to the increment method at invocation time and only returning the ownership when the method exits, would enforce sequential access to the counter, which does not correctly specify the behavior of a lock-free counter implementation. Some of the more recent specification techniques [2] use the concept of *atomicity* to broaden the behavior that can be captured.

Atomicity describes that an operation happens in a single, discrete instant in time, without possible interference from other threads. This property allows one to express a part of the behavior of hardware instructions such as compare-and-swap which is not captured by resource disjointness. However, atomicity is still not a good fit for more complex operations such as operations on data structures, because in those cases the operations are typically not atomic. Furthermore, it is often enough to know that a set of operations appear to behave atomically relative to each other. We describe this concept as *abstract atomicity*.

As illustrated by the concurrent lock-free counter, the concept of resource disjointness is in general not expressive enough to specify the behavior of concurrent code. Hence, to properly specify and verify lock-free concurrent code such as a concurrent lock-free counter, we require a logic that goes beyond the concept of resource disjointness, for example, a logic that successfully incorporates the concept of abstract atomicity which describes a useful property that is not captured by resource disjointness. Unfortunately, tool support for such logics is rare, which makes it difficult and time

consuming to apply such logics to larger examples. This, in turn, makes it difficult for researchers to experiment with these logics, e.g. to investigate their strengths and weaknesses, and for practitioners to apply them to real-world systems.

TaDA [2] (Time and Data Abstraction) is such a logic that combines the benefits of abstract atomicity and resource disjointness. Our goal is to provide tool support for TaDA. Therefore, we develop Voila, a tool that takes as input a source program enriched with TaDA-specific annotations and outputs if the verification was successful and, in case of failure, emits meaningful error messages. Our design choice is to find a good tradeoff between a fully-automatic verifier, where the tool tries to search for the correctness proof on its own, and a declarative proof checker, where the tool only checks the validity of a correctness proof given by the user.

Furthermore, we do not build the tool from scratch, but build it on top of the Viper intermediate verification language [8], a verification infrastructure for permission-based reasoning. This choice is motivated by Viper's claim to provide a backend that simplifies the development of more complex, permission-based verifiers. Part of this thesis will be to evaluate this claim.

To summarize, this work has two goals: the main goal is to develop, implement, and evaluate Voila (Core goal 1-3), a verifier for lock-free data structures and algorithms that is based on the TaDA logic. The secondary goal is to evaluate to which extent Viper suffices as a back-end for encoding TaDA (Extension goal 4).

Core Goals

1. Understand the TaDA logic and consolidate Voila's current features

The first goal is to understand the TaDA logic itself [2] [1] [3], to get an overview of the current features of the already existing Voila prototype, and to solidify the current state of the Voila prototype. This includes experimenting with the logic, testing the currently supported features, and rounding out existing features, e.g. checking transitive and reflexive closedness of action relations, checking the stability of region interpretations, and adding indexed and counting guard algebras.

2. Develop and implement the encoding of additional TaDA features

The second goal is composed of the following three steps: First, getting an overview of all available TaDA features. Second, prioritizing the collected features. Finally, developing and implementing the encoding of the features with the highest priority. Features with a low priority are added as an extension goal. We distinguish between high priority and low priority features because covering all of them would exceed the scope of this thesis.

Features are prioritized by analyzing the impact of the feature on soundness, set of verifiable programs, and usability of the verification language. For example, features that are required to ensure soundness have the highest priority and features that would facilitate certain proofs have a lower priority. An initial set of features is given below, ordered by their estimated priority.

- **Region levels:** In TaDA, *regions* are used to define the behavior of shared memory resources, e.g. a concurrent counter. *Levels* are assigned to such regions to prevent unsound cyclic reasoning. Therefore, region levels have a high priority.
- **View shifts:** Verifying programs includes modifying auxiliary state, e.g. creating new regions. In the logic of TaDA, those modifications are justified by *view shifts*. One way to enable such modifications in Voila is to let the users encode them into low-level instructions on their own. Because this approach makes it difficult for the user of the tool to guarantee soundness, a more systematic solution is needed. View shifts have a high priority because of their impact on soundness.
- **Thread-local resources:** In concurrent algorithms, threads can hold local memory resources, i.e. memory which is not accessible by other threads. For example, in a program that concurrently filters the elements of a list, each thread, at some point in time, may store

filtered elements locally. TaDA is able to specify such *thread-local resources* and additionally can capture more complicated actions such as publishing local memory resources to other threads. The impact of thread-local resources on the set of verifiable programs and on the usability of the tool are yet unclear and so is its priority.

- **Total-TaDA:** *Termination* of concurrent programs cannot be verified within basic TaDA itself; Total-TaDA [3] is an extension of TaDA that enables such total-correctness proofs. Because Total-TaDA is an advanced and complex addition on top of TaDA, we assign it a low priority.

3. Evaluate Voila

The third goal is to evaluate expressiveness, performance, and usability of the Voila tool. For expressiveness, examples have to be collected. On one hand, programs that were not verifiable without specific newly-added features, e.g. a ticket lock that requires an indexed guard algebra. On the other hand, programs that allow a more flexible specification together with a more realistic implementation, which was not possible without certain newly-added features.

For performance, we plan to evaluate the running time of the verifier on input programs of different sizes. Furthermore, we are interested in example programs whose verification exhibits performance issues and how these performance issues depend on the use of particular Voila features.

For usability, we plan to measure the annotation overhead in different programs and classify it according to the TaDA features that cause it.

Extensions

1. Soundness of Voila

One extension goal is to sketch a soundness argument of the Voila tool. A complete soundness proof would probably exceed the scope of this thesis.

2. Develop and implement the encoding of low-priority TaDA features

As mentioned in the second core goal, we will prioritize available TaDA features. Developing and implementing the encoding of TaDA features with a low priority is an extension goal.

3. Investigate limitations of Voila

An extension goal it to evaluate the limitations of Voila. One possible way to do this might be to collect examples that appear to not be verifiable in Voila and to determine the cause for that, i.e. if Voila does not support a certain feature, or otherwise, if they cannot be verified in TaDA at all.

The examples could be collected from other logics' research papers [5] [6] [9] [10] and existing tools [4], or created by us. A particular interest lies in examples that utilize more complex concurrent programming concepts such as helping and speculation.

4. Extending Viper

Part of the goal of this work is to investigate whether or not Viper is a suitable backend for more complex verification languages such as TaDA. For TaDA features that cannot, or only partially or with bad performance, be implemented on top of Viper, an extension goal is to investigate possible Viper extensions and to analyze their importance for the Voila tool.

5. Reducing annotation overhead

As mentioned before our design choice is to find a good tradeoff between a fully-automatic verifier and a declarative proof checker. A possible way to increase the degree of automation and to reduce the amount of required annotations might be to use syntax-driven heuristics. In order to be useful, such heuristics should not introduce performance issues. One extension goal is to investigate,

implement, and evaluate such a heuristic. Evaluation can be done by measuring the change in the proof size and the running time of the verifier.

References

[1] Pedro da Rocha Pinto. *Reasoning with time and data abstractions*. PhD thesis, Ph. D. thesis, Imperial College London, 2017.

[2] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In *European Conference on Object-Oriented Programming*, pages 207–231. Springer, 2014.

[3] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. Modular termination verification for non-blocking concurrency. In *European Symposium on Programming Languages and Systems*, pages 176–201. Springer, 2016.

[4] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. Caper: Automatic verification for fine-grained concurrency. In *European Symposium on Programming*, pages 420–447. Springer, 2017.

[5] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *European Conference on Object-Oriented Programming*, pages 504–528. Springer, 2010.

[6] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *ACM SIGPLAN Notices*, volume 50, pages 637–650. ACM, 2015.

[7] Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *ACM SIGPLAN Notices*, volume 48, pages 561–574. ACM, 2013.

[8] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016.

[9] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *European Symposium on Programming Languages and Systems*, pages 290–310. Springer, 2014.

[10] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *European Symposium on Programming Languages and Systems*, pages 333–358. Springer, 2015.