Approaches to Thread-Modular Static Analysis[*]

Vesal Vojdani, Kalmer Apinis, and Simmo Saan

Institute of Computer Science, University of Tartu, Estonia
{vesal,kalmera,ssaan}@ut.ee

1 Introduction

As multi-threaded operating systems are increasingly used in safety-critical settings, verification and analysis techniques have had to adapt. Different approaches to analyzing concurrent programs have been proposed, but due to differences in formalism, it is not clear how they relate. For now, we focus on (abstract) denotational semantics of concurrency via nested fixpoints [Ferrara, 2008]. We will reformulate the specific analysis of Miné [2012] in constraint-based form. We preserve the key ideas, but make some simplifications for the sake of clarity. In particular, we use the same abstract domain for the analysis as well as the communication between threads.

The use of explicit constraint system variables for thread communication makes the ideas more clear; in particular, the parallels with rely-guarantee reasoning is transparent in our formulation. There are, however, also practical advantages (as well as drawbacks) to constraint-based static analysis. Mainly, as more intermediate values are preserved, it is easier to proceed with incremental analysis. The drawback is increased memory consumption.

2 Concrete Thread-Modular Semantics

We consider programs as flow graphs: a set of nodes N and edges E of the form (u, s, v) where u and v are the program points in N and s is an elementary instruction, i.e., either an assignment or a conditional guard. There is also a dedicated elementary edge that initializes the program. These instructions manipulate sets of program states, each state being an environment mapping from variables to integers; this set we denote with $\rho \in D = \mathcal{P}(V \to \mathbb{N})$. We assume the semantics of an instruction s is given by $[\![s]\!] \colon D \to D$. We describe the set of states ρ_u that reach a program point u as the least solution to the following constraint system:

$$\rho_u \supseteq [\![s]\!]\rho_v \qquad\qquad (u, s, v) \in E$$

In the concurrent setting, we assume we are given a set of threads $\mathcal{T}$ and each $t \in \mathcal{T}$ has its own flow graph (N_t, E_t). The concrete semantics of a (sequentially consistent) concurrent program can be given as the non-deterministic execution of all thread interleavings, but here we immediately give a more thread-modular, yet concrete, view of interleaving semantics suggested in a subsequent paper by Miné [2014]. We encode the control state within the mappings ρ by having dedicated program counter variables pc_t, so that $\rho(\mathrm{pc}_t)$ returns the (unique ID in $\mathbb{N}$ associated with) the program point in N_t that thread t will execute next. We can then apply $[\![s]\!]$ to these extended states, leaving the program counters untouched. The essence of the thread-modular view is that for each thread t, the effect of its execution is given by a function $I_t \colon D \to D$. With this function, the constraint system for concurrent programs remains almost

*This work was supported by the Estonian Research Council grant PSG61.

unchanged:

$$\rho_v \supseteq [\![s]\!]\rho_u \qquad (u,s,v) \in E_t \tag{1}$$
$$\rho_u \supseteq \{\sigma \in I_{t'}(\rho_u) \mid \sigma(\mathrm{pc}_t) = u\} \qquad u \in N_t,\ t' \neq t \tag{2}$$

In addition to local steps (1), we take into account the effect of other threads at each program point (2). The second constraint relies on the program counter variables to restrict influences. Given a complete description of the interference from other threads, the above system thus computes the interleaving semantics. Now, this interference can also be successively built by adding the following constraint:

$$I_t \supseteq \{(\sigma, \sigma') \mid \sigma \in \rho_u, \sigma' \in [\![s]\!]\rho_v\} \qquad (u,s,v) \in E_t \tag{3}$$

The idea is to solve this in a nested fashion by first stabilizing the first two equations and then propagating the effect to other threads. This nesting is more evident in the denotational form [Miné, 2014], but the key point is that we can now understand different abstractions of concurrent communication by considering more abstract representations of the thread-interference space $D \to D$.

3 Interference Abstraction and Global Invariants

In this abstract, we will consider only the abstraction used in the original paper [Miné, 2012]. This is obtained by flow-insensitive and non-relational abstraction of communications. Thus, control variables are forgotten and thread effects are collapsed to variables mapping to their updated values, i.e., $V \to \mathcal{P}(\mathbb{N})$. Using this abstraction, we can write the constraints more informatively by using auxillary constraint variables R and G that highlight the correspondence to rely-guarantee reasoning:

$$\rho_u \sqsupseteq R_t \qquad u \in N_t \tag{4}$$
$$G_t(x) \sqsupseteq \rho_u(x) \qquad (\_, s, u) \in E_t,\ x \in \mathsf{write}_s \tag{5}$$
$$\rho_v \sqsupseteq [\![s]\!]\rho_u \qquad (u,s,v) \in E_t \tag{6}$$
$$R_t \sqsupseteq G_{t'} \qquad t' \neq t \tag{7}$$

Each thread t can rely on what is guaranteed by all other threads, as expressed by the last constraint and enforced upon each program point by the first. The guarantee of a thread is computed by the second constraint, which only takes into account the values written at a given program point. This system is sound and no less precise than the formulation by Miné [2012], which applies the thread influences lazily; however, we can show that whenever expressions are evaluated, the results will coincide.

4 Scheduling Abstraction and Privatization

For priorities and mutexes, we follow the scheduler model of Miné [2012]. There are a fixed number of threads with unique priorities. The scheduler runs the thread with highest priority that is *ready*. We have lock, unlock, and yield instructions that can make threads not *ready*. We have previously proposed a constraint-based formulation based on privatization [Vojdani et al., 2016] for a simpler concurrency model. We track for each variable x, the set of mutexes always held whenever accessing x. This set is stored in $\Lambda(x)$. We further assume a sound must-lockset analysis is given that handles locks and unlocks, such that the set of locks definitely held at a given program point u is given by $\mathsf{locks}(\rho_u)$. We can then treat shared variables protected by locks as thread-local within critical sections. Let us attempt to use privatization to express the scheduling-sensitive analysis of Miné [2012] in constraint-based form.

$$
\begin{aligned}
\Lambda(x) &\subseteq \mathsf{locks}(\rho_u) && (\_, s, u) \in E_t, x \in \mathsf{write}_s && (8)\\
\rho_u(x) &\sqsupseteq R_t(x) && u \in N_t,\ \mathsf{locks}(\rho_u) \cap \Lambda(x) = \varnothing && (9)\\
G_t(x) &\sqsupseteq \rho_u(x) && (\_, s, u) \in E_t,\ x \in \mathsf{write}_s,\ \mathsf{locks}(\rho_u) \cap \Lambda(x) = \varnothing && (10)\\
\rho_v &\sqsupseteq [\![s]\!]\rho_u && (u, s, v) \in E_t && (11)\\
R_t &\sqsupseteq G_{t'} && t' > t && (12)\\
\rho_v &\sqsupseteq G_{t'} && (u, s, v) \in E_t,\ s \in \{\texttt{yield}, \texttt{lock}\},\ t' \neq t && (13)
\end{aligned}
$$

This system is sound with respect to the concrete scheduling semantics of Miné [2012]; however, his abstract semantics is more precise as thread communication is propagated in and out of critical sections in a pair-wise fashion. If two threads protect communication via a common lock, but a third thread does not, we would no longer privatize this variable, thereby computing influences between all threads. A more faithful representation of the interference-based approach would be possible if we compute Rely-Guarantee invariants depending on the sets of held mutexes. This remains as future work.

On the other hand, we are more precise by only consider flow-insensitive influence from higher-priority threads, except at yield and locking instructions where a lower-priority thread can potentially influence a higher-priority thread. It is not clear to us why Miné [2012] ignores priorities in influences.

5 Weak Memory Consistency

These flow-insensitive abstractions are fairly imprecise with respect to the sequentially consistent semantics. Since many architectures provide weaker consistency guarantees, flow-insensitive abstractions are a good starting point for analyzing programs running on modern architectures. We previously claimed that our multithreaded analysis is sound with respect to the Linux kernel's memory model Vojdani et al. [2016], but we provided no rigorous proof. Suzanne and Miné [2018] have considered TSO and PSO models more rigorously. Our goal is to compare these abstractions as well as partial-order reduction techniques from model-checking for the analysis of concurrent programs communicating over weakly consistent memory.

References

P. Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In B. Beckert and R. Hähnle, editors, *Proc. of 2nd Int. Conf. on Tests and Proofs, TAP 2008*, volume 4966 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2008. doi: 10.1007/978-3-540-79124-9_9.

A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Log. Methods Comput. Sci.*, 8(1), article 26, 2012. doi: 10.2168/lmcs-8(1:26)2012

A. Miné. Relational thread-modular static value analysis by abstract interpretation. In K. McMillan and X. Rival, editors, *Proc. of 15th Int. Conf. on Verification, Model Checking, and*

Abstract Interpretation, VMCAI 2014, volume 8318 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 2014. doi: 10.1007/978-3-642-54013-4_3

T. Suzanne and A. Miné. Relational thread-modular abstract interpretation under relaxed memory models. In S. Ryu, editor, *Proc. of 16th Asian Symp. on Programming Languages and Systems, APLAS 2018*, volume 11275 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2018. doi: 10.1007/978-3-030-02768-1_6

V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler. Static race detection for device drivers: The Goblint approach. In *Proc. of 31st ACM/IEEE Int. Conf. on Automated Software Engineering, ASE 2016*, pages 391–402. ACM, 2016. Press. doi: 10.1145/2970276.2970337