Recognition of Real-World State-Based Synchronization

Martin Wittiger and Timm Felden
University of Stuttgart, Institute of Software Technology

Abstract

In the real world, safety-critical embedded systems use state-based synchronization to avoid data races. Using constraint solving to tackle state, we have improved upon existing static data race analysis.

1 Introduction

Data races form a class of programming errors. They are notoriously difficult to find by software testing, yet cause severe problems. Safety-critical embedded systems rely on functional correctness that can only be achieved in the absence of data races. This domain, therefore, particularly requires tools to mitigate the risks in this area. Therefore, the development of static analysis tools, which can either detect data races or prove their absence, is a well-established research problem.

Whenever multiple tasks of a concurrent system access shared resources such as communication variables, software developers use synchronization patterns. Desktop software mostly relies on mutexes and monitors. Embedded systems avoid these two patterns. In our experience, almost all embedded systems employ interrupt enable/disable patterns to synchronize tasks. Static analysis tools handle these patterns easily and efficiently. In addition, state-based synchronization is prevalent. It is often hand-crafted to fit a specific system and typically relies heavily on scheduling properties such as task priorities.

State-based synchronization is generally hard to deal with using static analysis. Keul [1] performs an analysis step called simple path exclusion that recognizes state machines used for synchronization. Schwarz et al. [2] use the term flag-based synchronization to denote essentially the same thing. Both approaches only recognize simple patterns. They both pose strict, virtually syntactically verifiable requirements on variables forming state machines.

2 State Analysis

Our goal is to classify data races using constraint solving. Consider the following two small examples. Both examples involve a state variable `s` that we assume to be of an atomic type and thus free of data races and a shared variable `d` that might be subject to a data race. Greek letters are used to denote tasks.

In the first example, we we show a variation of the ordinary state-based locking. We want new states to be added easily, thus we use an *untaken* state 0 that is not owned by any thread. For the sake of simplicity, we restrict ourselves to two tasks with one access each. The resulting state machine is depicted in figure 1. A pseudo-C implementation is shown in figure 2.

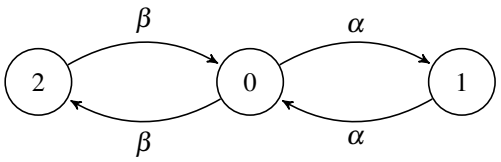

Figure 1: State transitions in our first state machine.

Task α	Task β
`if(s == 0) {`	`if(s == 0) {`
`  s = 1;`	`  s = 2;`
`  d = 4;`	`  d = 7;`
`  s = 0; }`	`  s = 0; }`

Figure 2: A state machine with two tasks and three states.

The second example (figure 3) features three tasks and a slightly degenerated state machine.

We will discuss whether the examples contain data races after taking a closer look at our approach.

2.1 Tool Composition

The analyses described in this paper build upon several layers of preexisting analyses. Our toolchain processes safety-critical C-code. In a first step, a pointer analysis, refined by escape analysis, is performed on the AST. We then perform a lockset analysis and establish a flow-, thread- and partly context-sensitive call graph using a project-specific concurrency configuration. An in-depth description is given by Keul [1]. All analyses are designed to be conservative, i. e. there will be a *warning* for each data race. Warnings are pairs of accesses to memory locations. In practice, many of them are false positives that we want to be able to safely exclude from the list.

Task α	Task β	Task γ
`if(s == 1) {`	`if(s == 2) {`	`s++;`
`  d *= 17;`	`  d += 37;`	
`  s = d; }`	`  s = d; }`	

Figure 3: A state machine with three tasks and multiple states.

This work was in part funded within the project ARAMiS by the German Federal Ministry for Education and Research with the funding ID 01IS11035. The responsibility for the content remains with the authors.

Afterwards, we perform dead code elimination and run a constant folding and propagation analysis. This is very important since constants used in synchronization mechanisms in real code are rarely just integer literals.

In the next stage, state variables have to be selected. This step is discussed in the next section as it has to be performed manually at the moment. Now, the call graph is projected on the effects on the selected state variables. This means that each statement in every basic block in every procedure in each thread is replaced by a conservative approximation of its effect. For instance, if a statement provably cannot change the value of any state variable, it is replaced by a `nop`.

When assigning to a dereference, for instance `*p = 7`, one cannot in general know, which memory location is affected. Such an assigment may thus be transformed into several weak updates on any state variable `p` may point to. However, if `p` points to a specific memory location, a single (strong) update is produced whenever this is a state variable or otherwise a `nop` is emitted. We treat function pointers in a similar fashion. So, the straightforward usage of pointer analysis leads to a natural translation that conservatively approximates pointers. As a whole this takes a form of abstract interpretation.

The projection also retains and marks conflicting accesses from warnings. The projection result including those data race marks is then transferred to a constraint language. The solver then discards all warnings whose accesses can be shown not to be concurrently reachable.

In our implementation we use CSP_M (machine-readable CSP, see [3]) as a constraint language and the FDR2 [4] refinement checker. Certain peculiarities of FDR2 require us to preprocess the CSP_M output of our tool before passing it on. When processing small examples like the ones shown in this paper, FDR2 answers instantly. Solver runtime presumably becomes an issue in more complex settings.

2.2 Choice of State Variables

We do not consider all variables eligible to represent state. To remain conservative, we have identified five criteria:

- Accesses to state variables must be atomic. This can be achieved in several ways: By declaring them atomic, by disabling certain compiler optimizations and using appropriate types, or by suitably enabling and disabling interrupts. This atomicity implies the absence of superfluous assignments and data races.
- State variables must be declared `volatile`. The C memory models do not provide strong enough guarantees about the visibility of updates to non-volatile variables to base synchronization on.
- Any state variable must be compared to a constant in a path predicate at least once. If they are not, they cannot possibly provide a means of synchronization.
- A variable that is never assigned a constant should not be used as it probably is of no use.
- State variables should be communication variables, i. e. they should be accesible from more than one task. Any variable local to a single task is likely to contribute little or nothing to synchronization.

Our tool rejects proposed state variables if they appear to be involved in data races and warns the user when non-volatile variables are used.

2.3 Examples Unveiled

When we use our prototype to examine the example code from figure 2, it refuses to eliminate the warning on the variable `d` seemingly protected by state-based synchronization. This is simply because the synchronization is broken—indeed, figure 1 is intentionally misleading.

State machines are safe when each state is owned by at most one task. States that have multiple owners will typically yield data races, while states with no owner result in deadlocks, as they cannot be left. Here, `s == 0` allows both tasks to access the shared resource and introducing priorities would not change anything.

The second example is tricky: It seems that Task γ allows for uncoordinated state changes breaking the state pattern. If, however, the priority of γ is lower than that of both α and β, the mechanism works and there is no data race. If γ has the highest priority, there is a data race on `d`. Our tool reports both cases correctly and is not misled by the curious assignments of `d` to `s` nor by the wrap-around semantics of `s`.

The real-life embedded C code we have encountered inspires this example. The ability to recognize this pattern leads to the elimination of false positives, which in turn saves QA staff time.

3 Conclusion and Future Work

Judging whether a given implementation of state-based synchronization works as intended is difficult and error-prone. We are confident that in future our approach will scale to industry-sized programs and dispense with manual identification of variables. Our work provides a means to verify the correctness of a given implementation. We improve upon existing solutions by reducing the requirements on state variables. We have demonstrated how our constraint-solving approach works on small programs and enables programmers to verify that synchronization patterns work as intended.

References

[1] S. Keul, "Tuning Static Data Race Analysis for Automotive Control Software," in *11th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2011, pp. 45–54.

[2] M. D. Schwarz, H. Seidl, V. Vojdani, and K. Apinis, "Precise Analysis of Value-Dependent Synchronization in Priority Scheduled Programs," in *LNCS*, vol. 8318, 2014, pp. 21–38.

[3] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall International, 2004.

[4] Formal Systems (Europe) and Oxford University, "Failures-Divergence Refinement: FDR2 User Manual," 2010.