Clara: a framework for implementing hybrid typestate analyses

Technical Report Clara-2

Eric Bodden

Darmstadt University of Technology [⋆]
bodden@cs.tu-darmstadt.de

Abstract. We present CLARA, a novel static-analysis framework for the implementation of hybrid static/dynamic typestate analyses. CLARA uses static typestate analyses to automatically convert any AspectJ monitoring aspect into a residual runtime monitor that only monitors events triggered by program locations that the analyses failed to prove safe. If the static analysis succeeds on all locations, this gives strong static guarantees. If not, the efficient residual runtime monitor is guaranteed to capture property violations at runtime.
Researchers can easily integrate their own static typestate analyses into CLARA. We instantiated CLARA with three static typestate analyses and applied these analyses to monitoring aspects generated from tracematches and by the JavaMOP runtime-monitoring tool.
CLARA is available as open source. We hope that other researchers will soon be joining us in using CLARA, and that this will foster progress in the field of typestate analysis.

1 Introduction

A typestate property describes which operations are available on a group of inter-related objects, depending on this group's internal state, the typestate. Software engineers use typestate properties to describe important properties of the programs they develop. For instance, a program should not write to a connection handle while this handle is in state "closed". Figure 1 shows a finite-state machine that expresses the language of all program executions that violate this property. Researchers have developed static [3,5,6,8,9] and dynamic [1,7] analysis tools that, given a program and a set of typestate properties that this program should adhere to, verify whether the program may violate these properties. While static-analysis tools inspect the program's code, dynamic analysis tools instrument the program under test with a runtime monitor that will notify the user of any property violations that do occur at runtime. Many dynamic analysis tools use a two-staged approach to instrument the program under test. The tools generate instrumentation code in the form of AspectJ aspects. The

[⋆] At the time at which this research was conducted, Eric was a Ph.D. candidate at McGill University, under supervision of Laurie Hendren.

user can then enable runtime checks for the program under test by weaving these aspects into the program. That way, AspectJ has become a popular intermediate representation for runtime-monitoring tools.

In this work we present Clara, a novel framework that aids researchers in implementing hybrid typestate analyses, i.e. static analysis with a dynamic monitoring component. Clara uses static typestate analyses to automatically convert any AspectJ monitoring aspect into a residual runtime monitor that only monitors events triggered by program locations that the analyses failed to prove safe. If the analysis succeeds on all program locations then this is a strong static guarantee that the given program cannot violate its typestate properties. Otherwise, the residual runtime monitor is guaranteed to catch property violations that do occur at runtime. Because this monitor is only notified about events at the program locations that the static analysis failed to prove safe (often just a few), the residually instrumented program usually executes much faster than a fully instrumented program would execute.

2 The Clara framework

Figure 2 gives an overview of Clara. The major design goal of Clara was to allow researchers to combine a wide range of static typestate analyses with a wide range of runtime monitors. To achieve this goal, we needed to de-couple the runtime-monitoring efforts from the static-analysis components. Because, as we mentioned, AspectJ has become a very popular intermediate representation for runtime monitors, we decided to use this same layer of abstraction for Clara's input as well. Hence, with Clara, the researcher first defines a set of runtime monitors for typestate properties using a runtime-monitoring tool. Normally, this tool would then generate a plain-AspectJ aspect. However, to communicate the typestate properties themselves to Clara's static analyses the tool also generates a special annotation to the aspect, called a "dependency state machine". It is easy to extend the monitoring tools to generate these annotations and we have modified JavaMOP and the tracematches compiler accordingly. If the monitoring tool does not support generating annotations the researcher can also easily write the annotation by hand.

As Figure 2 shows, Clara first weaves the monitoring aspect into the program. The dependency state machine defined in the annotation provides Clara with enough domain-specific knowledge to then analyze the woven program. Researchers can add a number of static analyses to Clara and have them applied in any order. When any of these analyses determines that an instrumentation point is irrelevant to all stated properties, i.e. can neither lead to a violation of this property, nor can prevent a property violation, then Clara disables the instrumentation at this point. The result is an optimized instrumented program that updates the runtime monitor only at locations that remain enabled.

In addition, users can instruct Clara to modify the advice dispatch of the monitoring aspect in such a way that the program under test can be used for Collaborative Runtime Verification [4]. In Collaborative Runtime Verification,

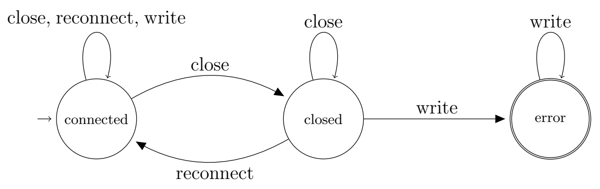

Fig. 1. "ConnectionClosed" example typestate property

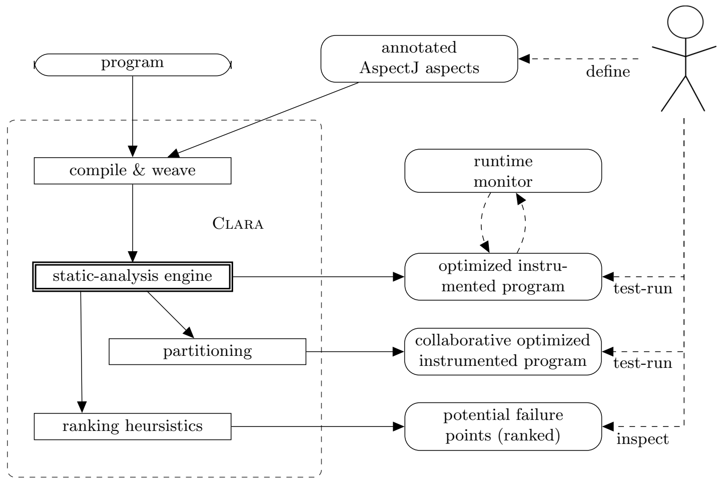
program
annotated AspectJ aspects
define
compile & weave
Clara
static-analysis engine
partitioning
ranking heursistics
runtime monitor
optimized instrumented program
test-run
collaborative optimized instrumented program
test-run
potential failure points (ranked)
inspect

Fig. 2. Overview of Clara

different users are sent differently configured versions of the program under test, where each version only contains partial monitoring code. This usually helps keep the monitoring overhead low. By design, this partitioning of instrumentation points is orthogonal to the static-analysis engine, i.e. it can be used in combination with any static analysis (or all of them).

3 Why researchers should use Clara

We believe that Clara is an attractive framework to researchers in the fields of both dynamic and static typestate analysis alike. Researchers who experiment with dynamic typestate analysis through runtime monitoring can use Clara's static analyses to obtain more efficient runtime monitors. For some monitoring approaches, converting a "full" runtime monitor into a residual one can cause speed-ups of several orders of magnitude [5,6] and can therefore make monitoring approaches realistic that seem unrealistic without such static-analysis component. In our experiments, we have seen such speedups with runtime monitors generated from tracematches [1] and JavaMOP [7]. Making a runtime-monitoring tool fit for Clara is often easy. Many such tools already generate AspectJ aspects anyway. Augmenting the aspect with a dependency state machine is often an easy task. Clara then handles everything else automatically.

Clara is also an attractive framework for researchers in the field of static typestate analyses. The typestate-analysis problem is generally undecidable and therefore any static typestate analysis must have some amount of false positives, i.e. the analysis will report that the given program may violate its typestate properties at program locations at which no violations will actually occur. A purely static analysis would just require the user to deal with these false alarms. Using Clara, the analysis can be used to automatically specialize an appropriate runtime monitor that only signals property violations as they actually occur at runtime. Further, Clara offers convenient abstractions and templates to implement static typestate analyses. We provide Clara pre-equipped with three example analyses of varying precision and complexity [5,6].

Clara is freely available as open source at http://bodden.de/clara/, along with extensive documentation, the author's dissertation, which describes Clara in detail, and with benchmark results. We hope that other researchers will soon be joining us in using Clara, and that this will foster progress in the field of hybrid static/dynamic program analysis.

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding Trace Matching with Free Variables to AspectJ. In: OOPSLA. pp. 345–364. ACM Press (Oct 2005)
2. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: An extensible AspectJ compiler. In: AOSD. pp. 87–98. ACM Press (Mar 2005)
3. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: OOPSLA. pp. 301–320 (Oct 2007)
4. Bodden, E., Hendren, L., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. Journal of Logics and Computation (Nov 2008), doi:10.1093/logcom/exn077
5. Bodden, E., Hendren, L.J., Lhoták, O.: A staged static program analysis to improve the performance of runtime monitoring. In: European Conference on Object-Oriented Programming (ECOOP). LNCS, vol. 4609, pp. 525–549. Springer (2007)
6. Bodden, E., Lam, P., Hendren, L.: Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time. In: Symposium on the Foundations of Software Engineering (FSE). pp. 36–47. ACM Press (Nov 2008)
7. Chen, F., Roşu, G.: MOP: an efficient and generic runtime verification framework. In: OOPSLA. pp. 569–588. ACM Press (Oct 2007)
8. DeLine, R., Fähndrich, M.: Typestates for objects. In: European Conference on Object-Oriented Programming (ECOOP). LNCS, vol. 3086, pp. 465–490. Springer (Jun 2004)
9. Dwyer, M.B., Purandare, R.: Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In: International Conference on Automated Software Engineering (ASE). pp. 124–133. ACM Press (May 2007)