Clara: Partially Evaluating Runtime Monitors at Compile Time[⋆]
Tutorial supplement

Eric Bodden[1] and Patrick Lam[2]

[1] Technische Universität Darmstadt, Germany
[2] University of Waterloo, Ontario, Canada
eric.bodden@cased.de

Abstract. Clara is a novel static-analysis framework for partially evaluating finite-state runtime monitors at compile time. 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 use Clara with most runtime-monitoring tools that implement monitors as AspectJ aspects. In this tutorial supplement, we provide references to related reading material that will allow the reader to obtain in-depth knowledge about the context in which Clara can be applied and about the techniques that underlie the Clara framework.

1 Introduction

It is challenging to implement runtime-verification tools that are expressive, nevertheless induce only little runtime overhead. It is now widely accepted that, to be expressive enough, runtime-verification tools must be able to track the monitoring state of different objects or even combinations of objects separately. Maintaining these states at runtime is costly, especially when the program under test executes monitored events frequently.

Even worse, to be reasonably confident that a program does not violate the monitored property, programmers must monitor many different program runs. The more code locations a program contains at which the program may violate the monitored property, the more test cases one may need to execute to appropriately cover all possible execution paths through these code locations. Paired with potentially slow runtime monitors, this goal may be hard if not impractical to achieve.

We therefore developed the Clara [9] framework to partially evaluate runtime monitors at compile time. Partial evaluation brings two main benefits:

[⋆] This work was supported by CASED (www.cased.de).

1. The partially evaluated monitors usually induce a much smaller runtime overhead than monitors that are fully evaluated at runtime.
2. The partial evaluation can drastically reduce the number of code locations that one needs to consider when looking for code that may cause a property violation. This helps programmers to tell apart useful from useless test cases.

As we show in our accompanying research paper [17], Clara's partial-evaluation algorithms can often prove that a given program can never violate the monitored property. In these cases, monitoring becomes entirely obsolete.

Clara was designed such that it poses minimal restrictions on the runtime-verification tool that generates the runtime monitor. Clara works with virtually all tools that generate runtime monitors in the form of AspectJ aspects.

In this paper we recapitulate Clara's architecture, explain its major design decisions and give pointers to further in-depth reading material.

2 Architecture of Clara

Clara targets two audiences: researchers in (1) runtime verification and (2) static typestate analysis. Clara defines clear interfaces to allow the two communities to productively interact. Developers of runtime verification tools simply generate AspectJ aspects annotated with semantic meaning, in the form of so-called "Dependency State Machines". Static analysis designers can then create techniques to reason about the annotated aspects, independent of the monitor's implementation strategy.

Figure 1 gives an overview of Clara. A software engineer first defines (top right of figure) finite-state properties of interest, in some finite-state formalism for runtime monitoring, such as Extended Regular Expressions or Linear-Temporal Logic, e.g. using JavaMOP [18] or tracematches [1]. The engineer then uses some specification compiler such as JavaMOP or the AspectBench Compiler [4] (abc) to automatically translate these finite-state-property definitions into AspectJ monitoring aspects. These aspects may already be annotated with appropriate Dependency State Machines: we extended abc to generate annotations automatically when transforming tracematches into AspectJ aspects. Other tools, such as JavaMOP, should also be easy to extend to generate these annotations. If the specification compiler does not yet support Dependency State Machines, the programmer can easily annotate the generated aspects by hand.

Clara then takes the resulting annotated monitoring aspects and a program as input. 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 analyze the woven program. The accompanying research paper [17] summarizes Clara's predefined analyses; further details can be found in previous work [10, 11, 14] and the first author's dissertation [9]. The result is an optimized instrumented program that updates the runtime monitor at fewer locations. Sometimes, Clara optimizes away all updates, which proves that the program cannot violate the monitored property.

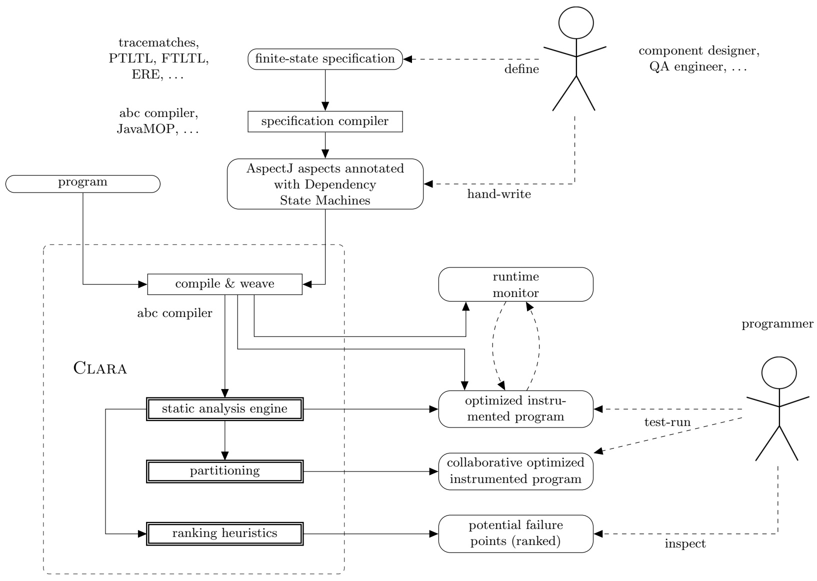

Fig. 1: Overview of CLARA

In addition, CLARA supports Collaborative Runtime Verification [13]. In Collaborative Runtime Verification, users execute differently-configured versions of the program under test; each version only contains partial monitoring code. Collaborative Runtime Verification interacts smoothly with the static analyses.

Finally, CLARA includes a set of built-in ranking heuristics [15]. These heuristics rank all program points that CLARA reports as "potential point of failure" according to a computed confidence value. This confidence value enables CLARA to prioritize program points where the program most likely violates the stated typestate property. Program points at which a violation is still possible, but not likely, will show up further down the ranked list. In addition, CLARA associates with each potentially property-violating program point all other program points that may have contributed to this violation, enabling programmers to easily inspect the context of the violation.

CLARA is available as open-source software at `http://bodden.de/clara/`, along with extensive documentation, the first author's dissertation [9], which describes CLARA in detail, and benchmarks and benchmark results.

In the following sections we discuss further reading on CLARA, explain how CLARA relates to existing approaches to runtime monitoring and static typestate analysis.

3 Further reading on Clara and its analyses

CLARA started out as an extension to the AspectBench Compiler [4] that was specific to one single specification formalism for runtime monitors, called tracematches [1]. At ECOOP 2007, we presented a set of three static analyses that attempt to statically optimize tracematches at compile time [14]. The three analyses presented there are similar to the three analysis stages that CLARA contains today, however they were all bound to tracematches; they did not generalize to any other monitoring tool. Further, the third analysis stage from the ECOOP paper, the "Active-shadows Analysis", is entirely different from today's Nop Shadows Analysis. The former analysis did not work at all: too coarse-grained abstractions resulted in both bad performance and bad precision.

In 2007, we presented an approach to Collaborative Runtime Verification [12]. In this approach, runtime monitors are spread accross multiple users; every user only monitors a subset of the original instrumentation points. It is non-trivial to select subsets of instrumentation points that (1) still have the potential of causing, in combination, a property violation, and (2) will not cause any false warnings at runtime. In our approach, we present an algorithm to select such subsets. We further present an algorithm that enables certain subsets only from time to time. This trades recognition power for runtime: the program runs faster but may not detect all property violations. Our results showed that this approach scales very well. A journal version of this work appeared in 2008 [13]. CLARA contains an option to enable Collaborative Runtime Verification.

In 2008, we presented [15] a replacement for the ineffective Active-shadows Analysis. This new analysis improves on the Active-shadows Analysis:

- It uses intra-procedural must-alias information to allow for strong updates. In many situations it helps to know that two variables *must* point to the same object. Similarly, the new analysis now uses may-alias information that is flow-sensitive on the intra-procedural level (opposed to being flow-insensitive everywhere). We use a novel pointer abstraction, called Object Representatives [16], to transparently combine the different sources of alias information. The original Active-shadows Analysis had no access to such information, it only used flow-insensitive may-alias information.
- While the Active-shadows Analysis performed a flow-sensitive analysis of the entire program, the novel analysis inspects one method at a time. While the analysis analyzes this method flow-sensitively, it models outgoing method calls flow-insensitively. This trades precision for analysis time, speeding up the analysis significantly.

Further, we presented a novel ranking and filtering approach that aids programmers in finding "true warnings" in a set of potentially false warnings. For program points at which the static analyses issue a warning, the analyses collect information about possible sources of imprecision. If there are many such sources, then the warning is assigned a low probability of being a "true warning", otherwise a higher probability. The CLARA framework contains these filtering and ranking heuristics as well.

In 2009, in joint work with Feng Chen and Grigore Roşu [11], the developers of JavaMOP [18], we generalized the analyses from ECOOP so that they were applicable to AspectJ aspects in general, and to monitors generated by JavaMOP in particular. The analyses presented in this novel work are generalizations of the first two analysis stages from the ECOOP paper, however include also the following improvements:

- The Quick Check in [14] can only detect cases in which a monitor cannot reach a final state as a whole. The improved Quick Check from [11], on the other hand, considers individual paths to final states. This can yield advantages in case of complicated specifications.
- The Orphan Shadows Analysis in [11] is highly optimized. In [14], the analysis algorithm explicitly enumerated all possible combinations, i.e., subsets of instrumentation points. With 1000 or more points, there can be up to 2^{1000} subsets. While we only observed a few pathological cases where this exponential blow-up happened in practice, the novel implementation of the Orphan Shadows Analysis circumvents this problem through a new algorithm that requires no such enumeration.

Also in 2009, for the first author's dissertation [9], we extended the analysis approach to be a proper framework, Clara, that can be easily extended by others. For the first time, Clara provides a uniform way to (1) specify runtime monitors as annotated AspectJ aspects, and (2) integrate novel static typestate analyses. During the process, we discovered that the flow-sensitive analysis presented in 2008 [15] was incorrect: in certain cases it could occur that the analysis yielded optimized runtime monitors that give false warnings at runtime. (see [10] for an example) Interestingly, in the meantime Naeem and Lhoták had published [36] an improved version of our analysis from 2008 that contained the same mistake. In 2010, we published [10] a modified version of the analysis, called the Nop Shadows Analysis, which is the final version of the flow-sensitive analysis that Clara contains today. Opposed to the original analysis attempts, this new analysis now contains a backwards-analysis pass that computes for every instrumentation point information about all continuations of the control flow from this point. It was this crucial piece of information that the original analysis was missing. The first author's dissertation [9] proves this analysis (and the analyses [11] from 2009) sound.

4 Runtime monitoring tools

In the following we discuss a number of monitoring tools that influenced the design and implementation of Clara. We also discuss whether programmers could use these tools in combination with Clara.

4.1 Stolz and Huch

Our work was originally motivated by Stolz and Huch's work [38] on runtime-verifying concurrent Haskell programs. The authors specify program properties using linear-temporal-logic formulae. Such formulae are generally evaluated over a propositional event trace: a formula refers to a finite set of named propositions and any of the propositions can either hold or not hold at a given event. Stolz and

Huch implemented a runtime library that would generate a propositional event trace at runtime and update a linear-temporal-logic formula according to the monitored propositional values. The library reports a property violation when the formula reduces to **false**. The formulas that Stolz and Huch allow for can be parameterized by different values, similar to the object-to-variable bindings that Clara supports.

4.2 J-LO

We ourselves developed J-LO, the Java Logical Observer [8], a tool for runtime-checking temporal assertions in Java programs. J-LO follows Stolz and Huch's approach in large parts, however the propositions in J-LO's temporal-logic formulae carry AspectJ pointcuts as propositions. The J-LO tool accepts linear-temporal-logic formulae with AspectJ pointcuts as input, and generates plain AspectJ code by modifying an abstract syntax tree. J-LO extends the Aspect-Bench Compiler, which allows it to then subsequently weave the generated aspects into a program under test. Pointcuts in J-LO specifications can be parameterized by variable-to-object bindings. While the implementation of J-LO is effective in finding seeded errors in small example programs, it causes a runtime overhead that is too high to allow programmers to use J-LO on larger programs. Nevertheless, one could annotate the J-LO-generated aspects with dependency information and then use Clara's static analyses to remove some of this overhead.

4.3 Tracematches

Allan et al. [1] are the creators of tracematches. Tracematches share with J-LO the idea of generating a low-level AspectJ-based runtime monitor from a high-level specification that uses AspectJ pointcuts to denote events of interest. Nevertheless, the tracematch implementation generates runtime monitors that are far superior to those that J-LO generates. Avgustinov et al. [6] perform sophisticated static analyses of the tracematch-induced state machine to determine an optimal monitor implementation that satisfies three main goals:

1. The monitor implementation should be correct.
2. The monitor should allow parts of its internal state to be garbage-collected whenever possible without jeopardizing correctness.
3. The monitor should implement an indexing scheme that allows the monitor, at any event that binds a variable v to an object o, to quickly look up all state-machine instances for the binding $v = o$.

As Avgustinov et al. show, reclaiming memory (2) and indexing of partial matches (3) are both necessary to achieve a low runtime overhead in the general case. In all the experiments that we conducted with tracematches in our work, these optimizations were already enabled. Hence our experiments show that, while these optimizations are necessary, they may not always be sufficient on their own. However, in combination with Clara's analysis, the runtime overhead will be low in most cases. Another difference between Allan et al.'s analyses and ours is that Allan et al. only analyze the state machine, while we analyze both the state machine and the program. This allows us to disable instrumentation at program points where this is sound, hence making it easier to check the program for potential property violations already at compile time. Allan et al.'s analyses do not analyze or modify the program under test.

4.4 Tracecuts

Walker and Viggers developed tracecuts [41], an approach that monitors programs with respect to a specification given as a context-free grammar over AspectJ pointcuts. Context-free grammars are strictly more expressive than the finite-state patterns that we consider in Clara: the first author's dissertation [9, Chapter 2] shows that some properties exist that finite-state formalisms cannot express but that could be expressed as a context-free language. However, most interesting program properties are in fact finite-state properties.

It is unclear how much runtime overhead tracecuts induce. In previous work [5], we tried to compare the relative efficiency of J-LO, tracematches, tracecuts and another tool called PQL (see below). As we reported there, there is an implementation of tracecuts, but it is immature, and while its authors kindly gave us private access to their executables, they did not feel it was appropriate for us to use their prototype for our experiments.

4.5 JavaMOP

JavaMOP provides an extensible logic framework for specification formalisms [18]. Via logic plug-ins, one can easily add new logics into JavaMOP and then use these logics within specifications. As we already showed in this thesis, JavaMOP has several specification formalisms built-in, including extended regular expressions (ERE), past-time and future-time linear temporal logic (PTLTL/FTLTL), and context-free grammars. JavaMOP translates specifications into AspectJ aspects using the rewriting logic Maude [19]. JavaMOP aims to be a generic framework that should support multiple specification languages. Therefore, the designers of JavaMOP are careful when it comes to making assumptions about the specifications used with their framework.

To make JavaMOP compatible with Clara, Feng Chen extended [11] the JavaMOP implementation so that it would perform some limited analysis of the specification, so that JavaMOP could annotate the generated monitors with dependency information that Clara can use to partially evaluate these monitors at compile time.

4.6 PQL

The Program Query Language [35] by Martin at al. resembles tracematches in that it enables developers to specify properties of Java programs, where each property may bind free variables to runtime heap objects. PQL supports a richer specification language than tracematches: it uses stack automata rather than finite state machines, which yields a language slightly more expressive than context-free grammars. Martin et al. propose a flow-insensitive static-analysis approach to reduce the runtime overhead of monitoring programs with PQL. This approach inspired us to implement our Orphan Shadows Analysis. As the authors show and as we confirm in our work, such an analysis can be very effective in ruling out impossible matches. However, we also showed that a flow-sensitive analysis can yield additional optimization potential. PQL instruments the program under test manually, using the BCEL [20] bytecode engineering toolkit. If PQL used AspectJ instead, then is should be possible to optimize the generated monitor with Clara, similar to tracecuts. PQL was published as an open-source project, available for download at `http://pql.sourceforge.net/`. However, it appears that the project is no longer maintained.

4.7 PTQL

Goldsmith et al. [30] proposed PTQL, the Program Trace Query Language, which provides an SQL-like language for querying properties of program traces at runtime. The authors also provide "partiqle", a compiler for this language. The compiler instruments the program that is to be queried so that the program notifies monitoring code about the appropriate events at runtime. The monitor itself uses indexing trees to associate the monitor's internal state with the appropriate objects. It may be possible to evaluate parts of a program query at compile time, for instance when comparing a method name to a constant string. Partiqle resolves such parts of a query already during compilation. This is the same as the partial evaluation of pointcuts that happens in standard AspectJ compilers: these compilers also insert runtime checks only for parts of a pointcut that the compilers cannot determine at compile time. Partiqle resorts to a table-based approach to evaluate the remainder of the query at runtime. Because PTQL uses its own compiler, and is not based on AspectJ, one cannot currently use Clara to evaluate PTQL queries ahead of time. Even if PTQL did generate aspects for its monitoring needs, one would have to take into account that the PTQL language is very expressive and probably Turing complete. Hence it remains unclear whether one could effectively determine dependencies within a query at compile time, so that Clara could exploit these dependencies to optimize PTQL monitors.

4.8 Sub-alphabet sampling

Dwyer, Diep and Elbaum propose a novel mechanism to guaranteeing low runtime overhead even in the presence of multiple monitoring properties and in cases where programs need to update the internal state of monitors for these properties very frequently [23]. The authors first propose to combine multiple properties over objects of the same class into one large "integrated" property.

As the work shows, monitoring of this integrated property can be more efficient than monitoring of the individual original properties. Then second, the authors propose to project the monitor for this integrated property onto multiple sub-alphabet monitors, where each monitor monitors exactly one subset of the original alphabet Σ of events. These sub-alphabet monitors form a lattice that is isomorphic to the power-set lattice of Σ. By the way in which Dwyer et al. define their state-machine semantics, each individual monitoring automaton in this lattice is sound, i.e., cannot report any false positives. The authors show that programmers can gain fine-grained control over the perceived monitoring overhead by selecting a subset of monitors from the lattice. Further, the authors present several heuristics that attempt to select reasonable subsets automatically. As the results show, the sub-alphabet lattice allows for a flexible selection of monitors that gives programmers fine-grained control over their overhead. We therefore believe that the authors' technique is a valuable addition to our own efforts of reducing the runtime-monitoring overhead, in particular to Clara's component for Collaborative Runtime Verification.

4.9 QVM

Arnold, Vechev and Yahav present QVM, the "Quality Virtual Machine", an extension of IBM's J9 Java Virtual Machine that implements a set of techniques that aim at aiding programmers to debug their programs [3]. QVM comes equipped with support for virtual-machine-level monitoring of single-object typestate properties. Programmers can use a simple syntax to define typestate properties for any given Java class. QVM then instruments instances of such classes to track the instances' typestate at runtime. Once QVM detects and report that a typestate property was violated, it starts sampling method calls that the program issues on objects that are allocated at the same allocation site as the object for which the violation occurred. Naturally, the calling sequences for both objects are not necessarily the same. Yet, the authors argue that in most cases these sequences will be similar enough such that the sampled trace will help the programmers pinpoint the actual problem on the violating sequence and hence fix the bug in their program code. QVM's techniques are complementary to all of the static techniques that Clara provides and it would be interesting to integrate both tools into a common solution.

5 Typestate analysis

In the previous section we have described several approaches to runtime-verifying program properties through monitoring. Many of these properties are finite-state properties, i.e., one can express the properties using finite-state machines. In the scientific literature, there is a large body of work that attempts to determine finite-state properties of program already at compile time. In this literature, finite-state properties are often called typestate properties, and the related static analyses are called typestate analysis.

5.1 Typestate by Strom and Yemini

In their original paper on typestate [39], Strom and Yemini first describe the idea of having a value's type depend on an internal state, the typestate, associated with that value. Certain operations can change a value's type by transitioning from one typestate to another. Strom and Yemini used state charts [32] to describe the possible state transitions for a class of objects.

In the description by Strom and Yemini, typestate properties are restricted to describing the state of single objects. For example, their model does not allow the state of an iterator i to change when the iterator's collection c is modified. This is because the authors' model has no means of associating i with c. Recently, typestate properties have been enjoying renewed interest, and many current analyses, including ours, do support the analyses of such "generalized" typestate properties.

5.2 Fink et al.

Fink et al. present a static analysis of typestate properties [26]. Their approach, like ours, uses a staged analysis which starts with a flow-insensitive pointer-based analysis, followed by flow-sensitive checkers. The authors' analyses allow only for specifications that reason about a single object at a time. This prevents programmers from expressing multi-object properties such as FailSafeIter. Like us, Fink et al. aim to verify properties fully statically. However, our approach nevertheless provides specialized instrumentation and recovery code, while their approach only emits a compile-time warning. Also, CLARA supports a range of input languages so that developers can conveniently specify the properties to be verified, while Fink et al. do not say how developers might specify their properties.

5.3 Bierhoff and Aldrich

Bierhoff and Aldrich [7] recently presented an intra-procedural approach that enables the checking of typestate properties in the presence of aliasing. The authors' approach aims at being modular, and therefore abstains from potentially expensive whole-program analyses like the ones that CLARA uses. To be able to reason about aliases nevertheless, Bierhoff and Aldrich associate references with special access permissions. Their abstraction is based on linear logic, and using access permissions it can relate the states of one object (e.g. an iterator) with the state of another object (e.g. a collection). These permissions classify how many other references to the same object may exist, and which operations the type system allows on these references. The authors use reference counters to reclaim permissions to help their type system to accept more valid programs. In their approach, they assume that every method is annotated with information about how access permissions and typestates change when this method is executed. Of course this does not necessarily imply that it has to be the programmer who adds these annotations. Many approaches exist [2,25,27–29,31,33,34,37,42] that can infer program properties. Some can even infer typestate properties. All of these tools operate under the assumption that programs are "mostly correct": by observing mostly correct program runs, the tools can infer which behavior is "usual". Deviations from this usual behavior can then be encoded as typestate properties.

In comparison to Fink et al., Bierhoff and Aldrich's approach has the advantage of being modular: given appropriate annotations it can analyze any method, class or package on its own. Clara on the other hand needs the whole program to be present, and in particular expects a complete but nevertheless sufficiently precise call graph. When the whole program is available, and can be analyzed, then Clara gives programmers the advantage that it does not require any program annotations. Clara only requires annotations that describe error situations, not the program, and then automatically analyzes the program to see whether such error situations can occur. We have found that worst-case assumptions coupled with coarse-grained side-effect information are surprisingly effective.

Bierhoff and Aldrich define typestate properties via a textual representation of statecharts. Hence, programmers can conveniently model behavioral subtyping, as in the original typestate-checking methodology that Strom and Yemini proposed.

Because Bierhoff and Aldrich's work defines a type system and not a static checker like Clara, the workflow that a programmer has to follow in Bierhoff and Aldrich's approach is slightly different than it is in the case of using Clara. Clara allows the programmer to define a program that may violate the given safety property. Clara then tries to verify that the program is correct, and when this verification fails it delays further checks until runtime. Bierhoff and Aldrich's approach defines a type checker, and hence the idea is that the programmer is prevented from compiling a potentially property-violating program in the first place. This gives the advantage of strong static guarantees. After all, if the program does compile then the programmer knows that the program must fulfill the stated property. On the other hand, the type checker may reject useful programs that appear to violate the stated property but will not actually violate the property at runtime.

5.4 DeLine and Fähndrich

DeLine and Fähndrich's approach [21] is similar in flavor to Bierhoff and Aldrich's. The authors implemented their approach in the Fugue tool for specifying and checking typestates in .NET-based programs. Fugue checks typestate specifications statically, in the presence of aliasing. The authors present a programming model of typestates for objects with a sound modular checking algorithm. The programming model handles typical features of object-oriented programs such as down-casting, virtual dispatch, direct calls, and sub-classing. The model also permits subclasses to extend the interpretation of typestates and to introduce additional typestates, similar to the statecharts-based approach by Strom and Yemini. As in Bierhoff and Aldrich's approach, DeLine and Fähndrich assume that a programmer (or tool) has annotated the program under test with information about how calls to a method change the typestate of the objects that the method references. One fundamental difference between the two approaches is the treatment of aliasing. While Bierhoff and Aldrich used access permissions to reason about aliases, Fugue's type system tracks objects merely as "not aliased" or "maybe aliased". Objects typically remain "not aliased" as long as they are only referenced by the stack. The respective objects can change state only during this period. Once they become "maybe aliased", Fugue forbids any state-changing operations on these objects. This makes Fugue's type system less permissive than the system that Bierhoff and Aldrich describe: in the latter type system objects can change states even when they are aliased.

5.5 Dwyer and Purandare

Dwyer and Purandare use existing typestate analyses to specialize runtime monitors [24]. Their work identifies "safe regions" in the code using a simple static typestate analysis similar to [22]. Safe regions can be methods, single statements or compound statements (e.g. loops). A region is safe if its deterministic transition function does not drive the typestate automaton into a final state. A special case of a safe region would be a region that does not change the automaton's state at all. The authors call such a region an identity region. For regions that are safe but no identity regions, the authors summarize the effect of this region and change the program under test to update the typestate with the region's effects all at once when the region is entered, instead of at the individual shadows that the region contains. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. One possible disadvantage of such summary transitions may be that one loses the connection between the places in the code that perform a state transition and the places that actually cause these transitions. This makes it harder for programmers to investigate these program places manually to decide for themselves whether this part of the program could or could not violate the property at hand. Our static analysis does not attempt to determine regions; we instead decide if each single shadow is a nop-shadow. Dwyer and Purandare's analysis should be easily implementable in Clara and we encourage such an implementation.

6 Conclusion

In this work, we have described the general architecture of Clara and have given pointers to related work from the literature, work both by ourselves and others. Clara is available as open source at `http://bodden.de/clara/` and we encourage researchers to use it and extend it. The website also includes a mailing list, on which we will be happy to answer any questions that may arise.

Acknowledgements We thank everybody who contributed to the design and implementation of Clara, including the developers and maintainers of Soot [40] and abc [4], but in particular Laurie Hendren, Grigore Roşu, Feng Chen, Oege de Moor, Pavel Avgustinov, Julian Tibble, Ondřej Lhoták and Manu Sridharan.

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 (Oct 2005)
2. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Symposium on Principles of Programming Languages (POPL). pp. 4–16 (Jan 2002)
3. Arnold, M., Vechev, M., Yahav, E.: QVM: an efficient runtime for detecting defects in deployed systems. In: OOPSLA. pp. 143–162. ACM Press (2008)
4. 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 (Mar 2005)
5. Avgustinov, P., Tibble, J., Bodden, E., Lhoták, O., Hendren, L., de Moor, O., Ongkingco, N., Sittampalam, G.: Efficient trace monitoring. Tech. Rep. abc-2006-1 (March 2006), `http://www.aspectbench.org/`
6. Avgustinov, P., Tibble, J., de Moor, O.: Making trace monitors feasible. In: OOPSLA. pp. 589–608 (Oct 2007)
7. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: OOPSLA. pp. 301–320 (Oct 2007)
8. Bodden, E.: J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University (Nov 2005), `http://www.bodden.de/pubs/`
9. Bodden, E.: Verifying finite-state properties of large-scale programs. Ph.D. thesis, McGill University (Jun 2009), `http://www.bodden.de/pubs/`, available through ProQuest.
10. Bodden, E.: Efficient hybrid typestate analysis by determining continuation-equivalent states. In: ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. pp. 5–14. ACM, New York, NY, USA (2010)
11. Bodden, E., Chen, F., Roşu, G.: Dependent advice: A general approach to optimizing history-based aspects. In: AOSD. pp. 3–14 (Mar 2009)
12. Bodden, E., Hendren, L., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. In: 7th workshop on Runtime Verification at the 6th International Conference on Aspect-Oriented Software Development, Vancouver, Canada. LNCS, vol. 4839, pp. 22–37 (2007)
13. 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
14. Bodden, E., Hendren, L.J., Lhoták, O.: A staged static program analysis to improve the performance of runtime monitoring. In: ECOOP. LNCS, vol. 4609, pp. 525–549. Springer (2007)
15. 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 (Nov 2008)
16. Bodden, E., Lam, P., Hendren, L.: Object representatives: a uniform abstraction for pointer information. In: Visions of Computer Science - BCS International Academic Conference. British Computing Society (Sep 2008), `http://www.bcs.org/server.php?show=ConWebDoc.22982`

17. Bodden, E., Lam, P., Hendren, L.: Clara: a Framework for Statically Evaluating Finite-state Runtime Monitors. In: 1st International Conference on Runtime Verification (2010), in these proceedings.
18. Chen, F., Roşu, G.: MOP: an efficient and generic runtime verification framework. In: OOPSLA. pp. 569–588 (Oct 2007)
19. Clavel, M., Eker, S., Lincoln, P., Meseguer, J.: Principles of maude. Electronic Notes in Theoretical Computer Science (ENTCS) 4 (1996)
20. Dahm, M.: BCEL, `http://jakarta.apache.org/bcel`
21. DeLine, R., Fähndrich, M.: Typestates for objects. In: ECOOP. LNCS, vol. 3086, pp. 465–490. Springer (Jun 2004)
22. Dwyer, M.B., Clarke, L.A., Cobleigh, J.M., Naumovich, G.: Flow analysis for verifying properties of concurrent software systems. ACM Transactions of Software Engineering and Methodolology (TOSEM) 13(4), 359–430 (Oct 2004)
23. Dwyer, M.B., Diep, M., Elbaum, S.: Reducing the cost of path property monitoring through sampling. In: ASE. pp. 228–237. Washington, DC, USA (2008)
24. Dwyer, M.B., Purandare, R.: Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In: ASE. pp. 124–133 (May 2007)
25. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering (TSE) 27(2), 99–123 (Feb 2001)
26. Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. In: International Symposium on Software Testing and Analysis (ISSTA). pp. 133–144 (Jul 2006)
27. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe (FME). LNCS, vol. 2021, pp. 500–517. Springer (Mar 2001)
28. Gabel, M., Su, Z.: Javert: fully automatic mining of general temporal properties from dynamic traces. In: Symposium on the Foundations of Software Engineering (FSE). pp. 339–349 (Nov 2008)
29. Gabel, M., Su, Z.: Online inference and enforcement of temporal properties. In: ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. pp. 15–24. ACM, New York, NY, USA (2010)
30. Goldsmith, S., O'Callahan, R., Aiken, A.: Relational queries over program traces. In: OOPSLA. pp. 385–402 (Oct 2005)
31. Hangal, S., Lam, M.S.: Tracking down software bugs using automatic anomaly detection. In: ICSE. pp. 291–301 (May 2002)
32. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
33. Li, Z., Zhou, Y.: PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In: Symposium on the Foundations of Software Engineering (FSE). pp. 306–315 (Sep 2005)
34. Lo, D., Maoz, S.: Specification mining of symbolic scenario-based models. In: Workshop on Program analysis for software tools and engineering (PASTE). pp. 29–35 (Nov 2008)
35. Martin, M., Livshits, B., Lam, M.S.: Finding application errors using PQL: a program query language. In: OOPSLA. pp. 365–383 (Oct 2005)
36. Naeem, N.A., Lhoták, O.: Typestate-like analysis of multiple interacting objects. In: OOPSLA. pp. 347–366 (Oct 2008)

37. Pradel, M., Gross, T.R.: Automatic generation of object usage specifications from large method traces. In: ASE. pp. 371–382. IEEE Computer Society, Washington, DC, USA (2009)
38. Stolz, V., Huch, F.: Runtime verification of concurrent haskell programs. Electronic Notes in Theoretical Computer Science (ENTCS) 113, 201–216 (Jan 2005)
39. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering (TSE) 12(1), 157–171 (Jan 1986)
40. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: CASCON. p. 13. IBM Press (1999)
41. Walker, R., Viggers, K.: Implementing protocols via declarative event patterns. In: Symposium on the Foundations of Software Engineering (FSE). pp. 159–169 (Oct 2004)
42. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: Symposium on the Foundations of Software Engineering (FSE). pp. 35–44 (Sep 2007)