Efficient Trace Monitoring

Pavel Avgustinov[1], Julian Tibble[1], Eric Bodden[2], Ondřej Lhoták[3],
Laurie Hendren[2], Oege de Moor[1], Neil Ongkingco[1], Ganesh Sittampalam[1]
[1] Programming Tools Group [2] Sable Research Group [3] Programming Languages Group
University of Oxford McGill University University of Waterloo

Abstract

A wealth of recent research involves generating program monitors from declarative specifications. Doing this efficiently has proved challenging, and available implementations often produce infeasibly slow monitors. We demonstrate how to dramatically improve performance — typically reducing overheads to within an order of magnitude of the program's normal runtime.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Experimentation, Languages, Performance

Keywords Program monitoring, runtime verification, program analysis, aspect-oriented programming

1. INTRODUCTION

Generating program monitors from declarative trace specifications is currently a very active research area (*e.g.* [2–4]). Proposals have been put forward by the runtime verification and aspect-oriented programming communities. Both have discovered the difficulty of making trace monitoring feasible — clearly any naive implementation of an entity that observes the entire trace of program execution is bound to fail. The fact that many proposals remain at the stage of research prototypes and that there are few "real" implementations is a clear indication of the inherent difficulties.

We present our approach to declarative trace monitoring: tracematches. The novel contribution is the design of a trace monitoring feature with free variables, as well as an efficient, feasible implementation.

2. TRACEMATCHES

Figure 1 introduces our running example and illustrates the syntax of a tracematch. It shows a program monitor that checks the 'safe enumeration' property:

> After an enumeration is created, the datasource upon which it is based may not be modified while the enumeration is in use — that is, until the last call to its next() method.

The regular expression (line 15) picks out violations of this property. It matches the *filtered* execution history of a program at the point a violation occurs. This filtering removes all events which don't correspond to the alphabet of the regular expression (defined in lines 8–13).

```
1  pointcut vector_update () :
2      call(* Vector.add *(..)) || call(* Vector.clear ()) ||
3      call(* Vector.insertElementAt (..)) ||
4      call(* Vector.remove *(..)) ||
5      call(* Vector.retainAll (..)) || call(* Vector.set *(..));
6
7  tracematch(Vector ds, Enumeration e) {
8      sym create after returning(e) :
9          call(Enumeration+.new(..)) && args(ds);
10     sym next before :
11         call(Object Enumeration.nextElement()) && target(e);
12     sym update after :
13         vector_update() && target(ds);
14
15     create next* update+ next
16     {
17         throw new ConcurrentModificationException();
18     }
19 }
```

Figure 1. Tracematch for unsafe enumerators.

A complete semantics of tracematch matching is beyond the scope of this document, full details can be found in [1].

Our design differs from existing approaches in that tracematches allow free variables in symbols — the stipulation is that there must be a consistent binding of the free variables to program values to trigger a match; filtering, thus, becomes specific to possible values of the variables. For example, after filtering, the trace

create(ds1, e1) create(ds2,e2) update(ds1) next(e2) next(e1)

will match once with variable bindings (ds1,e1) — these can be accessed in the body. Using free variables, it is possible to query the history of specific object instances.

3. CHALLENGES

Efficiently implementing *any* trace monitoring feature is certainly no easy undertaking. We have identified a series of challenges that implementors must address, and propose solutions to them that have been proved to work in the case of tracematches.

The overall design of a trace monitor is similar to a parsing concern — we would like to recognise the interesting traces of the program, observing one event at a time. However, the usual textbook techniques for parser generation and optimisation apply only in a limited form, due to the presence of filtering and free variables.

Copyright is held by the author/owner(s).
OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

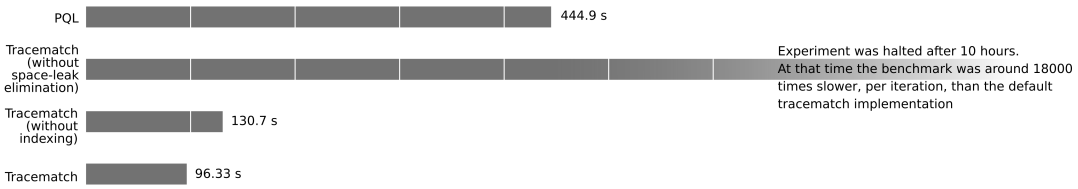

Figure 2. Performance comparisons

We choose to view the task of picking out relevant traces as a finite-state automaton recognising some language. This is a rather pervasive idea in the field, and much of the related work shares it.

CHALLENGE 1 (Automaton Construction). *Using automata to implement trace monitors is a natural idea, but some care must be taken.*

Both performance and correctness can suffer if this is approached naively. We identify the requirements and develop an algorithm for constructing a matching automaton from a tracematch.

CHALLENGE 2 (Partial Matches). *During program execution, some record of partially completed matches must be kept, since matching happens in an incremental fashion — the trace is observed one event at a time. Comparison to related work has shown that doing this naively by using some generic data structure incurs unnecessary overheads.*

Our solution is to generate partial match classes that are specialised to the particular tracematch. We represent the matching state in disjunctive normal form — each disjunct corresponds to a filtered trace that might lead to a complete match. By customising the representations of these disjuncts, we are able to reduce memory usage and ensure access time to bound variables is a simple field access. Also, a lot of the conditional logic required in the updates that happen when a new event is observed can be unrolled into the specialised methods, resulting in faster execution.

CHALLENGE 3 (Space Leaks). *Since tracematches allow the capture of program values, it is natural to be concerned about possible space leaks — if some object is captured by a TM variable, it might not be reclaimed by the garbage collector as it normally would have been, and over the course of a program execution memory will be wasted.*

We propose a comprehensive set of analyses to address this problem, categorising the free variables a tracematch defines into groups according to their memory behaviour, and eliminating space leaks whenever possible (while giving a compile-time warning when it isn't).

CHALLENGE 4 (Partial Match Representation).
A common problem is that even though a large number of partial matches can be accumulated during the execution of the program, each event only requires the update of a small subset of these.

We exhibit an algorithm and data structure (*indexing*) which significantly alleviate this problem — whenever possible, only the relevant partial matches are traversed during an update.

4. PERFORMANCE AND FUTURE WORK

Figure 2 shows the performance differences that the optimisations make. The comparison with PQL [3] gives an indication of the advantages of generating specialised code for representing automata and partial matches, because PQL is a similar trace monitoring system which does not perform those two optimisations.

We plan to develop additional analyses that would allow some tracematches to be evaluated statically — a user could thus opt for a significantly increased compilation time, but if a whole-program analysis can determine that certain tracematch-relevant events always (or never) occur, the instrumentation can be adapted to take this into account.

We believe there is also further scope to improve our indexing data structure and algorithm.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA '05*, pages 345–364. ACM Press, 2005.

[2] Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In *AOSD '04*, pages 141–150. Addison-Wesley, 2004.

[3] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: a program query language. In *OOPSLA '05*, pages 365–383. ACM Press, 2005.

[4] Robert Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *ACM Sigsoft International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159–169. ACM Press, 2004.