Making Trace Monitors Feasible

Pavel Avgustinov Julian Tibble Oege de Moor

Programming Tools Group, University of Oxford, UK

{pavel.avgustinov, julian.tibble, oege.de.moor}@comlab.ox.ac.uk

Abstract

A *trace monitor* observes an execution trace at runtime; when it recognises a specified sequence of events, the monitor runs extra code. In the aspect-oriented programming community, the idea originated as a generalisation of the advice-trigger mechanism: instead of matching on single events (joinpoints), one matches on a sequence of events. The runtime verification community has been investigating similar mechanisms for a number of years, specifying the event patterns in terms of temporal logic, and applying the monitors to hardware and software.

In recent years trace monitors have been adapted for use with mainstream object-oriented languages. In this setting, a crucial feature is to allow the programmer to quantify over groups of related objects when expressing the sequence of events to match. While many language proposals exist for allowing such features, until now no implementation had scalable performance: execution on all but very simple examples was infeasible.

This paper rectifies that situation, by identifying two optimisations for generating *feasible* trace monitors from declarative specifications of the relevant event pattern. We restrict ourselves to optimisations that do not have a significant impact on compile-time: they only analyse the event pattern, and not the monitored code itself.

The first optimisation is an important improvement over an earlier proposal in [2] to avoid space leaks. The second optimisation is a form of indexing for partial matches. Such indexing needs to be very carefully designed to avoid introducing new space leaks, and the resulting data structure is highly non-trivial.

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

General Terms Experimentation, Languages, Performance

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

Keywords Program monitors, runtime verification, program analysis

1. INTRODUCTION

Trace monitors observe the current execution trace, and execute some extra code when the trace matches a given pattern. Many runtime verification concerns can be expressed as trace monitors very naturally, simply by picking out violating traces.

A trace monitor is usually specified declaratively in two parts: firstly, a pattern describing which traces should match, and secondly, an action that should be executed when a program trace matches. The actual implementation of the trace monitor is automatically generated from its specification, typically in the form of instrumentation of a base program.

There is a very large amount of previous research on this topic, *e.g.* [2, 7, 8, 11, 13–16, 19, 21, 22, 25, 26, 28, 30]. These studies range from applications in medical image generation through business rules to theoretical investigations of the underlying calculus. The way the patterns are specified varies, and temporal logic, regular expressions and context-free languages have all been considered.

One theme shines through all of these previous works: trace monitors are an attractive, useful notion, worthy of integration into a mainstream programming language. This has not happened, however, because it turns out to be very difficult to generate efficient code when the trace monitor is phrased as a declarative specification.

The challenge is particularly severe when the specifications trace the behaviour of a group of several objects simultaneously. For example, when checking that a lock is always acquired and released in the same method invocation, we need to track the locked resource, as well as the method invocation in question. Similarly, when checking that a collection is not modified while iteration is in progress, we need to track the state of the collection as well as its iterators. For this reason, numerous of the above proposals (in particular [2, 11, 14, 21, 25]) have a facility for *binding* multiple variables during the matching process. Many more systems do not allow free variables in the trace patterns; in our experience, this feature is indispensable for many real-world examples, and therefore the focus of this work is optimising systems *with* free variables.

In our experiments, however, none of these systems (including our own in [2]) managed to generate *feasible* trace monitors. The generated monitors are adequate as a proof of concept, but they cannot be used in practice on substantial systems.

For some applications, it may be possible to use the substantial body of related work on *static* type-state verification, *e.g.* [18]. However, that invariably entails interprocedural analysis of the observed code. This is costly, and the assumption that all the code is available before load-time cannot be satisfied in practice. Furthermore, before such costly techniques are explored, we must first determine how far one can get with cheaper techniques.

Contribution This paper shows, for the first time, how to generate *feasible* trace monitors from declarative specifications that use free variables. Furthermore, our techniques rely only on analysis of the trace specification, not on costly whole-program analyses of the monitored code. Any monitoring system, regardless of the chosen specification formalism, must implement the techniques presented here to achieve feasibility.

Specifically, the detailed contributions are as follows:

Benchmarks:

- We present the first benchmark set of substantial, realistic applications of runtime monitoring.
- Each of these monitors has been coded by hand (in the programming language AspectJ), and also in the specification formalism of [2].
- This set of benchmarks (which is publicly available at [1]) thus provides the first solid basis for experimental evaluation of trace monitoring features.

Leak detection and prevention:

- We demonstrate that space leaks are a show-stopping bottleneck when naïvely generating trace monitors.
- In [2] an analysis was briefly sketched for eliminating that problem. We identify a crucial flaw in that analysis, and show how it can be remedied via the novel notion of *persistent weak references*. We then go on to show how to extend the analysis to detect and prevent additional leaks.
- Unlike [2], we then proceed to carefully evaluate the effects of the optimisation.

Indexing of partial solution sets:

- We exhibit another show-stopping performance problem, namely the need to update the set of partial solutions whenever a relevant event occurs.
- We propose an automatic technique for choosing an appropriate index structure on the set of partial solutions, which is purely based on the monitor specification.
- Furthermore, we present an indexing data structure that does not introduce new space leaks, and thus combines well with the above leak prevention technique. Again this involves very careful and subtle use of weak references: naïve indexing would worsen space leakage.
- Finally, we provide a set of algorithms to update such indexing data structures (avoiding costly unnecessary intermediate data-structures) and present an argument for why these algorithms are correct.

2. TRACE MONITORING

We first outline the basic strategy for generating executable code from trace monitors, via a specific example. Variations of the same code generation strategy can be found in any trace monitoring system, but our primary focus is on tracematches [2] — the system that this work evaluates and optimises.

As previously mentioned, a trace monitor is a combination of a pattern and an action to run when the program trace matches that pattern. In a tracematch, the pattern is defined in three parts: a set of variables, an alphabet of symbols (defined in terms of the variables), and a regular expression over the symbols.

A common runtime verification property is that of safe-enumeration:

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

To check this property with a tracematch, two variables are required. One of them will range over collections that might be iterated; in the Java 1.2 API these are instances of the class Vector, and so we will use the identifier v. The second variable, e, ranges over enumerations. Symbols are defined using AspectJ pointcuts — a language for intercepting runtime events [23] — but for clarity we give an informal definition of the three required symbols.

CREATE	an enumeration e is created from a Vector v
UPDATE	the Vector v is modified
NEXT	the nextElement() method is called on e

Finally, the regexp which specifies a violation of the safe-enumeration property is 'CREATE NEXT$*$ UPDATE$+$ NEXT'.

What does it mean for such a regular expression (which includes free variables) to match the program trace? Consider the execution history shown in Table 1. The left side of the table shows a sequence of symbol-matching events from a program trace. For each event, the corresponding symbol and the objects bound to *v* and *e* are shown.

On the right side of the table, we show two substitutions of objects for tracematch variables. Each substitution defines a projection of the original sequence to a string over the symbol alphabet, found by including the symbols from lines on the left of the table that have compatible variable bindings.

The tracematch action is triggered if there is some substitution for which the regular expression matches a suffix of the projected string. For example, in the table above, the substitution $v = v_1 \wedge e = e_1$ results in the projected string CREATE NEXT UPDATE NEXT, which matches the regular

Symbol	v	e	$v = v_2$ $e = e_3$	$v = v_1$ $e = e_1$	
CREATE	v_1	e_1		CREATE	
CREATE	v_1	e_2			
NEXT		e_2			
NEXT		e_1		NEXT	
UPDATE	v_1			UPDATE	
CREATE	v_2	e_3	CREATE		
NEXT		e_3	NEXT		
NEXT		e_1		NEXT	$(*)$
UPDATE	v_2		UPDATE		

Table 1. An example execution history

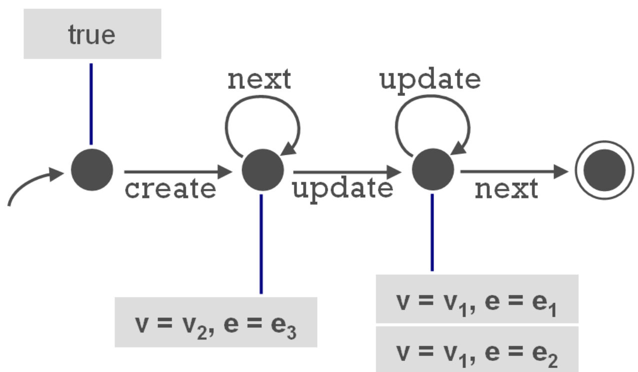

Figure 1. An automaton with states labelled by constraints

expression. The action is triggered *at the event* which completed the match (*i.e.* the line marked $(*)$, not the line after, even though the projected string still matches).

When a tracematch is compiled, the regular expression is translated into a finite state automaton. Such automata are well understood, and can be easily constructed from regular expressions. Variants also exist for other formalisms such as context-free grammars and linear temporal logic. In particular, automata are much better suited to *online* matching, which is why they are used here instead of keeping a regular expression representation.

As we saw above, matches with different variable bindings should be independent of each other and may be interleaved. This means that, conceptually at least, a separate finite automaton is required for every possible binding — an unbounded number. However, it is possible to simulate this collection of automata by using a single automaton and labelling its states with constraints. An example of such a labelled automaton for the safe-enumeration property is shown in Figure 1. A constraint C, labelling a state i, is interpreted as follows: "there is an automaton in state i for each variable binding that satisfies C". Constraints are boolean expressions, consisting of $(x = v)$, $(x \neq v)$, $\wedge$, and $\vee$ (for tracematch variables x and objects v), and are stored in disjunctive normal form.

The reader may be wondering, as is quite common in our experience, why we chose to label the states of the automaton with constraints. Would it not be simpler to store each variable binding together with a state counter and, when an event occurs, find the appropriate binding and just update the counter? There are two problems with this technique. Firstly, it is not applicable in cases where the implementation may store partial variable bindings (where only some of the monitor variables are bound to objects). Secondly, the fine-grained leak-elimination strategies described in Section 4 are not possible when bindings are stored in this way.

For the interested reader, the actual source for the safe-enumeration monitor is shown below, but the focus of this paper is code generation rather than language design.

```
1  tracematch(Vector v, Enumeration e) {
2      sym create_enum after returning(e) :
3        call (Enumeration+.new (..)) && args(v);
4      sym call_next before :
5        call (Object Enumeration.nextElement()) && target(e);
6      sym update_source after :
7        vector_update () && target(v);
8
9      create_enum call_next * update_source + call_next
10
11     {
12            throw new ConcurrentModificationException ();
13     }
14 }
```

Lines 2–7 declare the individual event patterns, Line 9 contains the regular pattern that is matched against the current trace, and Lines 11–13 constitute the extra code that is executed upon a successful match. A detailed discussion of the pertinent language design decisions can be found in [2].

3. BENCHMARKS

We now present our benchmark collection. In selecting benchmarks, our main inspiration was the literature on runtime verification, and also on trace-based aspect-oriented programming. We scrupulously included base programs where the overheads are likely to be substantial. An alternative, taken in [9], is to apply a single tracematch to many unrelated base programs. This allows research into cheap strategies for eliminating inapplicable monitors (indeed [9] demonstrates such a strategy). However, it is not appropriate in this context because it would wrongly give the impression that overheads are low, simply because the benchmarks may not heavily engage in the monitored events. Also, we provide a 'gold standard' for each benchmark, consisting of a hand-coded best possible solution.

To our knowledge, this is the first collection of trace monitoring benchmarks especially constructed to cover a wide variety of properties, while highlighting potential overheads. It is publicly available for others to use in comparative experiments. Indeed, the fact that the designers of JavaMOP recently adapted some of our examples for their own system [17] confirms that these benchmarks are not tailored just for tracematches, and lends support to our claim that they set a useful standard for all runtime verification systems.

Below we first informally describe the benchmarks, each consisting of a (monitor, base program) pair. Then we define the gold standard to compare against, namely highly hand-optimised AspectJ implementations. Finally we take our initial measurements, using the naïve implementation of tracematches to determine where the main performance problems are.

Monitors and base programs

SAFEENUM This is the example mentioned in the previous section: we would like to throw an exception when a collection is modified while an enumeration over that collection is in progress. We apply it to JHotDraw [20], a Java graphics package that allows the user to animate a drawing's components in a visually appealing manner.

The animation loop contains a call to `Thread.sleep` to slow down movement; we removed that to truly observe the overheads incurred by trace monitoring, and also minimised the window to factor out the cost of display operations. The effect of these measures is that enumerating the figure elements at each frame dominates the cost of the animation loop, and one would thus expect overheads to be very large indeed. JHotdraw's use of enumerations is actually unsafe because one can edit the drawing while animation is in progress, and when that is done, our monitor catches the violation.

NULLTRACK This is a debugging concern. We aim to provide a trace monitor that can help track down the cause for a NullPointerException. We do this by intercepting all field writes where a field f is set to **null**, followed by no intervening assignment of a non-**null** value until we see a read of f followed by a null pointer exception — the trace monitor then reports all fields that were set to **null**, and where that happened. However, to improve precision of the reported possible causes, we also demand that the field get that is assumed to have triggered the exception and the exception itself occur on the same line.

Note that this is intrinsically a very expensive monitor, because it involves a large number of instrumentation points: all field reads and writes, and all method calls. We applied it to CertRevSim, a discrete event simulator used to evaluate the performance of various certificate revocation schemes.

HASHCODE Another common runtime verification concern, this trace monitor checks the property that whenever an object is stored in a data structure indexed by its hash code (*e.g.* a HashSet), and subsequently a membership test is executed, the object's hash code hasn't changed in the meantime (if it has, we could get false negatives). This is expensive to check at runtime for a different reason: if the base program makes heavy use of hash-based data structures, the trace monitor will have to keep track of very many partial matches, one for each object stored.

We applied this to two different base programs: Firstly, AProVE [3], a termination prover for term-rewriting systems which we chose for its heavy use of hash sets, and also because it is precisely the type of complex, large application where this tracematch can be helpful. We also applied HASHCODE to Weka [31], a machine learning library that makes extensive use of HashMaps, but which is substantially smaller than AProVE.

OBSERVER The observer pattern is a popular example in the aspect-oriented programming community. Whenever a subject changes its state, all registered observers must be notified; with AOP (and with trace monitoring) it is possible to achieve this without the subject explicitly making such notifications.

The base program here is AJHotDraw [27], an aspect-oriented rewrite of JHotDraw, which uses the observer pattern for its display updates. Again, this benchmark has been chosen to fairly measure worst-case overheads: in AJHotDraw, each subject has precisely one observer; there is thus considerable cost involved in using a data structure that caters for multiple observers, without knowledge of the one-to-one correspondence. Again, we removed all delays from the animation loop and minimised the window to factor out display operation overheads.

DBPOOLING This is an example proposed by Laddad in his AspectJ text book [24]. The idea is that in a particular database application, creating and establishing connections is the most expensive operation, so, whenever possible, we want to pool existing connections and reuse them, rather than creating new ones.

The base program in this case is artificial — a slight modification of the example in Laddad's book, simply connecting to the database multiple times and performing database operations. The AspectJ version we used is Laddad's. Note that for this example, we expect the trace monitor to *improve* performance rather than hinder it, since it would prevent unnecessary connections from being established.

LUINMETH Here we are concerned with checking a stylistic rule. Whenever a lock is acquired, it should be released before the enclosing method returns. Checking this rule with a trace monitor was proposed as a motivating example by the authors of PQL [25]. It is particularly interesting because it requires matching method entry/exit pairs at runtime, which is in general a context-free language problem. However, making judicious use of variable bindings, it turns out that it is possible to express this using weaker formalisms; we encoded it in tracematches, which only allow regular expressions as patterns.

The base program we chose here is Jigsaw [29], the W3C's leading edge web server platform. It makes frequent use of locking and unlocking, so there are plenty of points of interest. In fact, Jigsaw violates the style rule being checked, and our monitor catches the violations.

REWEAVE The *abc* compiler makes use of an optimisation phase termed *reweaving*, during which the effects of weaving aspect code are undone and weaving is repeated using more precise analysis results obtained on the previously woven code [5]. Of course the correctness of this relies on properly undoing the effects of the first weaving step, so that the reweaving process can start form a clean state. In terms of trace monitoring, if a field is written to during the execution of the weave() method, is *not* written to during unweaving, and is read during reweaving, then it is very likely that it was not reset properly, and an error should be reported.

We applied this to the *abc* compiler itself, and used the instrumented version to compile a small program to obtain our numbers.

Gold standard AspectJ implementations We implemented the trace monitoring concern for each of these benchmarks as a tracematch, and also created a hand-coded, hand-optimised set of plain AspectJ aspects that manually achieve the goal. We did this in order to have a 'gold standard' for each benchmark, the best possible implementation an optimising compiler might aim for.

In all cases, the AspectJ code is much longer and more complex than the tracematch specification. For example, it often makes nifty use of weak references, and sometimes it makes additional assumptions not available to the compiler (for instance that all collections being enumerated have been created in user code and not in libraries). In short, the AspectJ version is what an expert programmer would do for the problem in hand.

Discussing the entire set of hand-coded solutions is beyond the scope of this paper, but we may highlight just one example in detail. The general idea is that we want to follow the "standard" method of ensuring fail-fast iterators (keeping modification counters), but without modifying the standard libraries. For that purpose, we subclass *Vector* to *MyVector*, which keeps a version number as a field, which gets incremented upon each update operation. That increment is implemented with a piece of advice; with another piece of advice, all constructor calls on *Vector* are replaced by constructor calls on *MyVector*. It is for the replacement of those constructor calls that we need to be sure all instances of *Vector* are created within JHotDraw and not in some library code. An automatic test for verifying this property would require costly interprocedural analysis of the monitored code.

Each implementation of *Enumeration* also has such a version number, copied from the underlying vector upon creation; this is introduced by the aspect as an intertype declaration on the *Enumeration* interface. Each time an enumeration step is taken, a piece of advice compares the version number of the enumeration with that of the underlying vector: when they are not equal, an exception is thrown. All pieces of advice need to be declared as **synchronized**.

Such an implementation of the concern relies on a deep understanding of the property, and it seems unlikely that automatic code synthesis would ever be able to obtain it from a declarative specification like a tracematch pattern.

Measurements The initial performance measurements are given in Table 2; all benchmarks were run on a Java HotSpot 1.5.0_11 VM running on Linux. For this set of measurements, we disabled most tracematch optimisations, simply allowing the implementation to perform its code generation. In particular, no attention is paid to potential space leaks or to organising partial match sets; this is the approach that seems to be prevalent in the field. Let us examine some of the trends exhibited by these numbers in detail.

We can see that the tracematch performance is very close to that of AspectJ in the case of DBPOOLING; this can be explained by the fact that the benchmark does expensive database processing that dominates the monitoring overheads. More surprisingly, performance is very good in the case of LUINMETH, which is applied to a significantly larger base program. Here, the explanation is in the tracematch pattern. Recall that it is intended to find occurrences of a lock being acquired and not released by the time the acquiring method returns. When a method that acquired a lock does return, therefore, it is clear that none of the associated partial matches will need further updates, and they are invalidated. There is no build-up of 'live' partial matches over the course of the benchmark.

Performance looks less promising for some of the other benchmarks. REWEAVE is more than 50% slower, which is actually a lot worse than it sounds, because the instrumentation only affects a small part of the benchmark execution (the reweaving cycle); HASHCODE/WEKA is more than two times slower, NULLTRACK is — unsurprisingly — hugely affected by the large amounts of instrumentation, taking more than 1000 times longer than equivalent AspectJ instrumentation, and SAFEENUM and HASHCODE/APROVE are completely infeasible for our generated trace monitors. Even though we let each of them run for several hours, they did not reach anywhere near the end of the respective computations. The huge slowdown was especially visible with SAFEENUM, applied to a JHotDraw animation: the amount of animation steps per second dropped very rapidly until it was taking more than 10 seconds per frame, and that time was still increasing.

These numbers alone should be sufficient to motivate the need for further optimisations.

ID	MONITOR	BASE	KSLOC	NONE	ASPECTJ	NOOPT
1	SAFEENUM	JHOTDRAW	9.5	3.1S	3.3S	>90M
2	NULLTRACK	CERTREVSIM	1.4	0.15S	0.6S	748S
3	HASHCODE	APROVE	438.7	345S	478S	>90M
4	HASHCODE	WEKA	9.9	2.6S	2.7S	5.5S
5	OBSERVER	AJHOTDRAW	21.1	4.2S	4.4S	9.2S
6	DBPOOLING	ARTIFICIAL	<0.1	30.5S	2.9S	3.3S
7	LUINMETH	JIGSAW	100.9	14.1S	18.0S	21.6S
8	REWEAVE	ABC	51.2	7.8S	8.0S	12.0S

Table 2. Benchmark runtimes: No instrumentation, hand-optimised AspectJ and naïve tracematches

4. MEMORY LEAK ELIMINATION

In [2], Allan *et al.* briefly sketch an analysis and code generation strategy to avoid introducing memory leaks into the system. They give one example to show its effectiveness (a version of SAFEENUM), but there is no proper experimental validation. Crucially, there is a subtle but important flaw in their proposal, which we shall correct below by introducing the novel notion of *persistent weak references*. We then expand that previous work by generalising the analysis to a larger class of memory leaks, allowing early invalidation of partial matches even in situations that previously would have leaked. It is interesting to note that the effectiveness of this optimisation on its own critically depends on the heap size of the Java virtual machine.

The overall aim is to enable garbage collection of partial matches that are guaranteed not to reach a final state in the automaton. Roughly, that can be achieved when 'completing the match' would require an extra event on an object that is already garbage-collected itself; but because that object has expired, the extra event cannot occur. We first describe the analysis required to detect that situation.

Crucial to our strategy is the concept of a *weak reference*. Normally, a reference to a runtime object prevents that object from being reclaimed by the garbage collector. Weak references allow the programmer to refer to an object without preventing its destruction. In Java, this takes the form of the special class java.lang.ref.WeakRef, whose constructor takes the referent object as an argument. WeakRef provides a method called **get**() — calling it will return the referent, if it still exists, and **null** otherwise.

Categorising references The analysis works on the finite state automaton generated from the regular expression of the tracematch. For each non-initial non-final state in that automaton, the free variables of the tracematch are divided into three categories:

collectableWeakRefs Variables that are bound on *every* path from the current state to a final state.

weakRefs Variables that are not used in the tracematch body and are not in the above set.

strongRefs Variables that are not in the above two sets.

As an example, consider the SAFEENUM tracematch presented above. It only has two non-initial non-final states (*cf.* Figure 1). From the first of these, we need to take both an UPDATE and a NEXT transition to reach the final state; UPDATE binds the v variable and NEXT binds e, so both of these are **collectableWeakRefs**. On the second state, we only need to see a NEXT to get to the final state, so the only **collectableWeakRefs** variable is e. Since v isn't used in the tracematch body, it is a **weakRef** — if it was, it would be classified as a **strongRef** on that state.

Exploiting the categorisation For variables in the first category, it is sufficient to keep weak references (*i.e.* references that do not prevent garbage collection) to the bound values, since we are guaranteed to bind them again before reaching a final state, and could keep a strong reference then (if necessary). Moreover, if one of these weak references expires, then we can discard the entire partial match, since it cannot possibly reach a final state — any path to a final state would have to bind the expired runtime value, which is impossible. This observation might well improve the memory behaviour of trace monitors, since it could reduce the number of live partial matches.

Allan *et al.* claim that for **weakRef** variables, we also only need to keep weak references. The reason is that even if the runtime object expires, it would not actually be used, and so keeping a strong reference would unnecessarily prevent its garbage collection. A reference to it is only kept for matching purposes. Note, however, that discarding partial matches when such a variable expires is not justified, since by definition we can reach a final state without necessarily binding it again. It turns out that this is an oversimplified view that can lead to not all matches being successfully completed; we will explain this in a moment.

Finally, variables that are not necessarily re-bound on every path to a final state and are used in the tracematch body must be kept alive; hence we need to keep strong references — such variables form the **strongRef** category.

Of course, there are certain tracematches which inherently *do* introduce space leaks, and this categorisation of free variables allows the compiler to issue a warning to that effect: if there exists a non-initial non-final state for which

collectableWeakRefs is empty, then partial matches in that state could conceivably accumulate to an unbounded number without ever being discarded, and a warning should be emitted. Such warnings are very helpful in practice, because it is easy to forget about performance when writing declarative monitor specifications.

Persistent weak references To see why the original proposed treatment of **weakRefs** is not sound, consider the following simple example: Suppose we have a tracematch with two symbols, A and B, and that A binds a tracematch variable x. The pattern is AB, and the tracematch body doesn't use x, so that it is a **weakRef**. Imagine at some point during program execution, we have the following constraint on x:

$$(x = v_1) \vee (x = v_2)$$

and that then both v_1 and v_2 expire. Weak references to expired objects return **null**, and so now the constraint becomes

$$(x = ?) \vee (x = ?)$$

that is, we cannot tell the two disjuncts apart any more. Thus, when we see another B event, the tracematch body would be run once instead of twice.

This small example shows that we need to treat weak references that do not invalidate their entire partial match specially: We need to be able to tell them apart even after they expire. We propose the concept of *persistent weak references*, as explained below, to address this issue.

The defining characteristic of a persistent weak reference should be that after its referent expires, calling **get**() returns not **null**, but some object that uniquely identifies the original referent. Moreover, all persistent weak references to the same object should return the same value after it has been garbage-collected.

It is not immediately obvious how one could achieve such behaviour, but our work on *indexing* (*cf.* Section 5) suggests an approach that works: Make use of *collectable key identity maps*. We proceed by defining PersistentWeakRef, a subclass of the standard WeakRef class which has no publicly visible constructors. Rather, it provides a static public method getRefFor(Object o) that can be used to create new references. This method maintains a static identity map m from runtime objects to associated instances of PersistentWeakRef; when called, it first checks if m already contains its parameter, and if so simply returns the associated value. Otherwise, it constructs a new PersistentWeakRef, records the correspondence in m and returns it. Effectively this ensures that only one persistent weak reference object is ever constructed for each runtime value. The map m has special handling for its keys: They are stored as weak references (so as not to prevent their garbage collection), and moreover when they expire, the associated key/value pairs are discarded. In this way, the memory used by the the PersistentWeakRef class is proportional to the number of *live* referents, that is, it doesn't introduce any memory leaks itself.

Finally, we need to define the behaviour of the **get**() method on our new class. This proceeds as follows: First of all, dispatch to the superclass. If the result is non-**null**, the object is still alive and we can simply return it. If the result is **null**, we can return **this** — that is, the PersistentWeakRef instance. This satisfies our two requirements above, as we can still tell apart weak references to expired objects, and references to the same runtime object will return the same value when it expires.

Collectable combinations of variables It is possible that a state may have no **collectableWeaks** but have one or more *combinations* of variables for which, if all the objects bound to these variables (by a single partial match) expire, the whole partial match may be discarded.

For example, the following pattern detects erroneous uses of the Reader and InputStream library classes. The pattern is taken from a benchmark by Bodden *et al*, in their paper on using whole-program static analysis to optimise tracematches [9].

$$\mathit{create}\ (\mathit{readR}|\mathit{readI})*\ (\mathit{closeR}|\mathit{closeI})+\ (\mathit{readR}|\mathit{readI})$$

The pattern uses two variables, r and i, for the Reader and InputStream, respectively. The *create* symbol is the only symbol which binds both variables. The variable r is also bound by *readR* and *closeR*, whilst i is bound by *readI* and *closeI*.

The combination of alternation and the inherent symmetry of the pattern means that neither of the variables are ever classified as **collectableWeakRefs**. Although the objects bound to r and i may be garbage collected, the data structure for each partial match will *never* be discarded unless the match is completed.

Evidently this is a preventable source of memory leaks: if, for a single partial match, both the object bound to r and the object bound to i expire, then it is impossible to complete the match, and thus it could be discarded.

We have extended the leak-elimination analysis to compute such combinations of variables. The extension adapts the technique, used at runtime, of labelling states with boolean expressions. We will refer to these expressions as *collecting constraints*. They consist of $\wedge$, $\vee$, and assertions x (for tracematches variables x). For example, a state labelled $x \vee (y \wedge z)$ means "a partial match on this state can safely be discarded if the object bound to x has expired or the objects bound to y and z have both expired."

Collecting constraints are calculated using the following formula, where $i \xrightarrow{a} j$ means there is a transition from state i to state j labelled with the symbol a, and bound_a is the set of tracematch variables bound by a.

$$\mathit{collectable}_i \quad = \quad \bigwedge_{i \xrightarrow{a} j} \left(\mathit{collectable}_j \vee \bigvee_{x \in \mathit{bound}_a} x \right)$$

Since there may be cycles in a tracematch automaton, the equation is solved for each state simultaneously by iteration to find the least-fixpoint.

Note that this analysis detects a strictly larger class of memory leaks than the previous analysis. Variables categorised as **collectableWeakRefs** by the previous analysis are exactly those which appear as singleton assertions (*i.e.* like x in $x \vee (y \wedge z)$) when the collecting constraint is expressed in disjunctive normal form.

Measurements Let us now examine the effects of these optimisations on our set of benchmarks. Note that the following numbers are taken with standard heap settings (*i.e.* a maximum heap of 64M). We can see that we have made substantial performance improvements in some cases: NULLTRACK runs more than ten times faster now, HASHCODE/WEKA runs 30% faster, REWEAVE has improved somewhat, and — most pleasingly — SAFEENUM has become feasible. Some of the other benchmarks show little change, however.

ID	ASPECTJ	LEAKELIM	NOOPT
1	3.3S	97S	>90M
2	0.6S	61.5S	748S
3	478S	>90M	>90M
4	2.7S	3.9S	5.5S
5	4.4S	9.4S	9.2S
6	2.9S	3.3S	3.3S
7	18.0S	21.4S	21.6S
8	8.0S	11.4S	12.0S

Table 3. Run times in seconds after leak elimination.

It is worth examining SAFEENUM a bit more closely, as it is the clearest winner of leak elimination. As stated earlier, the base program is JHotDraw, a Java figure editor. For the benchmark, its animation routines were modified by removing all delays, so that the figure elements are moved around the screen as fast as possible. Animation is implemented by enumerating the figure elements at each step, and moving each of them slightly; in total, 100000 animation steps are performed.

There is a crucial difference between this trace monitor and the one in our LUINMETH benchmark, which proved to have very low overheads even without leak elimination: SAFEENUM checks that the `next()` method is never called on an enumeration after the underlying collection has been updated; in particular, there is no single event after which we can be sure that a specific partial match will never lead to a successful match, so the number of potential matches grows unboundedly with base program execution. In LUINMETH, we could discard partial matches when returning from the associated method.

Thus, SAFEENUM is plagued by terrible memory performance, since it has to keep a steadily growing number of objects in memory; also, all of these must be updated after every relevant event, and this explains the huge slowdown we described in the previous section.

Consider now the effects of the space leak elimination on this. The enumeration object will be classified as one of the **collectableWeakRefs**; thus, it can still be garbage-collected, and when it is, all associated partial matches are discarded. Now, each enumeration expires when the associated animation step is completed — the following animation step creates a new one. So whenever we complete an animation step, we drop all associated matching state, which leads to the benchmark's becoming feasible. Moreover, rather than having unboundedly increasing memory behaviour, it exhibits practically constant memory usage very slightly above that of the uninstrumented program, as described by Allan *et al.* — at least if we force periodic garbage collector runs (*cf.* Figure 8). The situation with NULLTRACK is similar; here, too, we have a pattern with no definite "end" event which would allow us to discard partial matches, and therefore being able to reduce the amount of work as objects expire is absolutely crucial.

So, it seems that space leak elimination is indispensable for many applications. We have observed significant speedups after enabling the optimisation, particularly for "open-ended" trace monitors (*i.e.* those for which it is not possible to rule out a match's completion before program termination). Most notably, many *liveness* or *safety* properties, which are popular in the runtime verification community and assert that some good condition always holds or some bad sequence of events never occurs, are of this type. Without observing object garbage collection and invalidating partial matches based on that, such trace monitors would have to keep track of ever-growing sets of potential matches.

However, there is something highly unsatisfactory about the technique presented so far. The fact is that the impact of this optimisation critically depends on the performance of the garbage collector. In modern JVMs, garbage collections tend to happen when the available heap space is running low; concretely, this means that if we run the program with a smaller heap, we'll see more GC runs, and conversely there will be fewer runs on a larger heap. This leads to the somewhat paradoxical situation that a particular program can run more slowly if it is given more heap space.

Figure 2 illustrates precisely this effect, taking SAFEENUM as an example. With very small heaps, the garbage collector runs all the time, and this slows the benchmark down. The optimum heap size in this case seems to be around 10M — this strikes a good balance between eliminating invalidated partial matches and not taking up too much computation time. Beyond that, there is a roughly linear increase in execution time with heap size, as the garbage collector runs less and less frequently, and hence the effect of the optimisation is diminished more and more. Indeed, with a heap size of 1.7GB, SAFEENUM remains infeasible — it fails to terminate even after several hours.

It is worth mentioning at this point that all of our benchmarks that didn't show great improvements after this optimisation perform small enough computations that even with lots of live matching state in memory, the standard heap

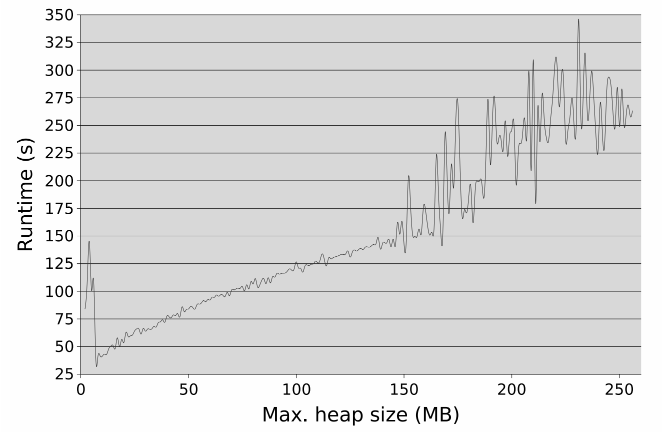

Figure 2. Runtime for SAFEENUM against heap size.

size is big enough. This is the reason why leak elimination seemed to have a small effect — the garbage collector isn't triggered often enough to make a significant difference. Running the benchmarks with a smaller heap size does show greater improvements, but of course ideally we want to have an implementation that doesn't behave *worse* with more memory.

Still, in our experience, the analysis of Allan *et al.* succeeds in eliminating many space leaks (when forcing regular garbage collector runs), and correctly emits warnings when there could be a leak. Moreover, the early cleanup of invalidated matching state leads to substantial performance gains in many situations. Our extension to the analysis allows us to handle even cases where the original strategy would have failed, particularly patterns making heavy use of alternations.

However, tracematch performance is still rather worse than that of equivalent AspectJ instrumentation, even in the best case. For SAFEENUM, the AspectJ version implements the usual technique for implementing safe iterators by putting logical time stamps on collections and iterators — it seems unlikely that this idea can be automatically synthesised from the specification.

The HASHCODE/APROVE (3) benchmark remains infeasible. Close examination of its behaviour reveals that space leaks have been eliminated, but all time is spent iterating over a large set of live partial matches. AProVE makes heavy use of hash sets, and each object stored in a hash set is potentially the source of a match completion. We will address this in the next section.

5. INDEXING

Recall that the basic implementation of trace monitors (as explained in Section 2) consists of a finite state machine where the states have been labelled with constraints; and that constraints are boolean combinations of variable bindings to objects.

The performance numbers shown so far were taken with a simple implementation that represents such constraints naïvely as sets of disjuncts. As we will see, large numbers of stored disjuncts will likely be irrelevant to any single update, but with a simple set representation every single one must be iterated over for each update. The overheads this causes make trace-monitoring infeasible for large classes of programs and monitors. This section details a data-structure for partitioning disjuncts, and algorithms for updating such structures, that avoid processing irrelevant disjuncts. The methods shown preserve matching behaviour and extend the techniques from Section 4 so that no new memory leaks are introduced.

To see what it means for a disjunct to be *irrelevant* we must summarise parts of the previous work on the semantics of tracematch matching [2]. A tracematch symbol is modelled as a function from events to constraints.

$$\mathit{symbol} = \mathit{event} \rightarrow \mathit{constraint}$$

For example, if a symbol a does not match the event e then $a(e) = \mathit{false}$, but if a does match the event e just in the case that the tracematch variable x is bound to the object o, then $a(e) = (x = o)$.

The set of all symbols declared by a tracematch is written A. We write $j \xrightarrow{a} i$ to mean there is a transition in the tracematch automaton from state j to state i that is labelled with the symbol a. For each state i, label_i denotes the constraint labelling it. When an event e occurs, the new label at each state i, written label'_i, is

$$\begin{aligned} \mathit{label}'_i \;\stackrel{\text{def}}{=}\; & \left(\bigvee_{j \xrightarrow{a} i} (\mathit{label}_j \wedge a(e)) \right) \\ & \vee \left(\mathit{label}_i \wedge \bigwedge_{a \in A} \neg a(e) \right) \end{aligned} \tag{1}$$

The first line of this equation says that if there is a partial match in state j, and the variable bindings for that partial match are compatible with $a(e)$, then that partial match can transition to state i. These are called positive updates. The second line states that some transition *must* be taken for each partial match, unless no symbol can be matched to e that would result in compatible bindings. These are called negative updates.

To illustrate, consider the safe-enumeration monitor from Section 2, together with the variable bindings shown in Figure 3. Suppose that a NEXT event occured with the variable binding ($e = e_2$). The calculations that should be performed to obtain the new constraints, in accordance with Equation 1, are shown in Figure 4. Indeed, when using a simple-set implementation, these calculations must be performed. However over half of them are redundant: note that three out of the five new disjuncts, when simplified, are either *false* or

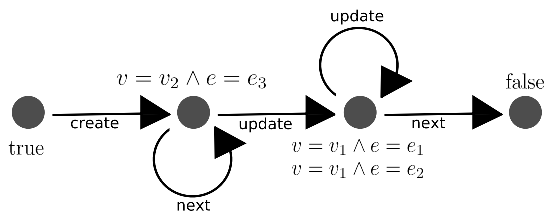

Figure 3. An annotated automaton for safe-enumeration.

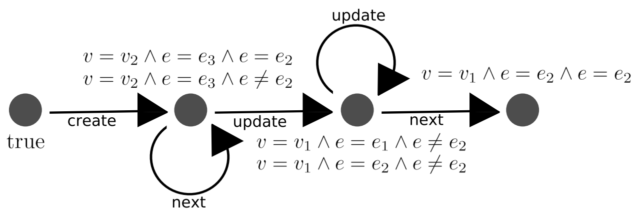

Figure 4. The calculations performed when using simple sets to store disjuncts.

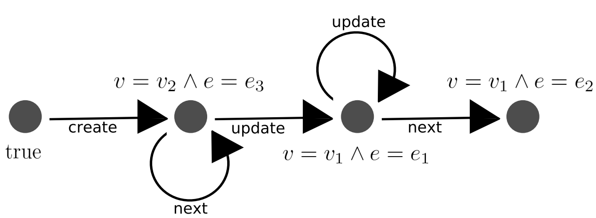

Figure 5. The new constraints after a NEXT event has occured.

unchanged — that is, a disjunct that is identical to one previously labelling the same state. These are the *irrelevant* disjuncts. The simplified version of the constraints is shown in Figure 5.

In general, suppose that the constraint labelling some state j has a disjunct d, which contains the equality $(x = o_1)$. If an event e occurs, and there is an a transition from j to i, we can see from the positive updates in Equation 1 that $d \wedge a(e)$ will be calculated as part of the new constraint labelling state i. Suppose, however, that the constraint generated by matching the event to the symbol a contains $(x = o_2)$ where $o_1 \neq o_2$. It is guaranteed that $d \wedge a(e) \equiv$ *false*, because it contains two contradictory constraints on x. The disjunct d is therefore irrelevant to a at the event e.

A similar situation is found when calculating negative updates. Suppose that state i is labelled with a disjunct $d = (x = o_1) \wedge d'$, and the same event e occurs such that $a(e) = (x = o_2) \wedge c$ (for some predicate c). Equation 1 shows that computing the negative updates for i will involve

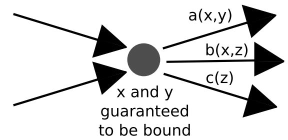

Figure 6. An example automaton state

calculating $d \wedge \neg a(e)$:

$$\begin{aligned}
d \wedge \neg a(e) &\equiv d \wedge \neg((x = o_2) \wedge c) \\
&\equiv d \wedge ((x \neq o_2) \vee \neg c) \\
&\equiv (d \wedge (x \neq o_2)) \vee (d \wedge \neg c) \\
&\equiv ((x = o_1) \wedge d' \wedge (x \neq o_2)) \vee (d \wedge \neg c) \\
&\equiv ((x = o_1) \wedge d') \vee (d \wedge \neg c) \\
&\equiv d \vee (d \wedge \neg c) \\
&\equiv d
\end{aligned}$$

In this case, d is also irrelevant for negative updates — not because it is falsified, but because d is *unchanged* after the update and continues to label state i.

The goal of indexing is to partition the disjuncts stored at each state so that as many irrelevant disjuncts are ignored as possible for each update.

5.1 Choosing a Partition

The tracematch implementation automatically chooses, for each state, a set of variables with which to partition the disjuncts stored at that state. For illustration, consider the state i, shown in Figure 6. Only the variables x and y are guaranteed to be bound and the state has three outgoing transitions labelled a, b, and c. The symbol a binds x and y, b binds x and z, and c binds just z. What variables should be used to partition disjuncts labelling i?

Firstly, a variable can only be used to partition disjuncts at a state if it is guaranteed to be bound at that state and is also bound by an outgoing transition. If this is not the case, then the definition of irrelevance shown above does not apply. There are therefore some transitions which cannot benefit from indexing; the c-transition on state i is such a transition because it only binds z, and z is not guaranteed to be bound at state i.

The strategy for choosing partition variables is to benefit as many outgoing transitions as possible, whilst ignoring those those that cannot benefit. More precisely, the set of partition variables is found by intersecting the variables that are guaranteed to be bound with the variables bound by each symbol that can benefit from indexing.

In the case of state i, the set is found by intersecting the set of variables that are guaranteed bound, $\{x, y\}$, with $\{x, y\}$ and $\{x, z\}$ (for a and b, respectively — c is not considered because it cannot benefit from partitioning). Therefore, the disjuncts at this state would be indexed by their binding for x.

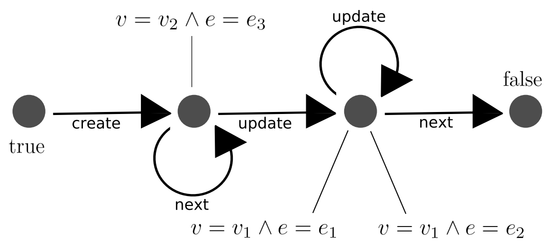

Figure 7. The safe-enumeration constraints, as stored using indexing.

It is possible that this method results in no partition at all because there are two or more mutually exclusive sets that could be partitioned on. The safe-enumeration monitor we have been considering is an example of this: the UPDATE event only binds the vector v, whilst the NEXT event only binds the enumeration e. Indeed, in general, there may be some examples where it is most performant to not partition the disjuncts at all. However, it is likely that the programmer will be able to judge which symbols are going to match the most often. For this reason, symbols may be marked as 'frequent' in a tracematch. If no partition can be chosen by the method described above then the process is repeated for just the 'frequent' symbols.

In the case of safe-enumeration, we marked the NEXT symbol as frequent, which meant that the variable e was chosen to index on.

5.2 A Data-Structure for Indexing

Partitions are represented by the tracematch implementation as trees, by using multiple levels of maps. Each level in the tree corresponds to a variable, and the map-keys at that level are objects bound to that variable. For safe-enumeration, the same constraints that appeared in Figure 3 are stored using indexing as shown in Figure 7. Writing the implementation of these maps requires careful effort in order not to break the optimisations of Section 4, because each map's keys are objects in the monitored program and the map must keep references to them.

The maps are specialised hash-tables that may keep weak references to the keys and can have extra code that is triggered when a weak reference expires. The behaviour upon a reference expiring differs, depending on the classification of the variable being indexed, as described above.

If the key variable is a **collectableWeakRef**, then every disjunct on the sub-tree indexed by that expired weak-reference also must have a collectable weak-reference to that garbage-collected object — it is therefore safe to drop the entire branch when the binding expires. Note that this is a particularly fast way of discarding invalidated constraints: rather than having to iterate over and check each one in turn, all constraints on this state with the same expired binding are dropped at once.

There is a potential problem, however, in that this could introduce a race condition. If an event occurs that does not benefit from the current index variable (as described above), we have to iterate over all key/value pairs at the current level in the indexing map. If the garbage collector invalidates one of the keys during this iteration, it will be removed from the map, and we have violated the Java API requirement of fail-fast iteration. The solution is a custom map implementation that knowingly deviates from the usual iterator contract. We allow *safe* modifications to the map during an iteration, and take care to ensure that dropping a key/value pair due to key expiry is safe in this sense.

In fact, there is one more pitfall of allowing the map implementation to discard branches at undefined times: One cannot rely on an iterator's hasNext() method to give the right result, since all remaining key/value pairs could conceivably be dropped before the call to next() even if hasNext() returned **true**. Thus, another contract modification is necessary: we allow next() to return **null** if there is no further element to iterate.

If the index variable is a **strongRef**, then no further care needs to be taken; we can use a simple identity hash map. Note that this is unlike the standard HashMap implementation, which considers keys to be equal subject to their equals() method — we really need object identity, due to the tracematch semantics.

Finally, let us consider the situation where the indexing variable is a **weakRef**, *i.e.* where we cannot invalidate constraints due to the variable being garbage-collected, but still need to keep a weak reference. Recall that this case proved especially tricky in Section 4. It may seem that indexing does not make sense for such variables. It is, however, still the case that constraints may benefit from indexing, at least while an object is still alive and there are events that bind it. Once it expires, we will never have to explicitly look it up in the map, but we need to keep the associated constraints accessible to iteration.

One approach might be to group together all key/value pairs with an expired key; as stated above, as long as we can iterate over them we can perform updates correctly. However that gives rise to rather unpleasant race conditions. When do we perform this grouping operation? Since a garbage collection can occur at any time, suppose one happened during an iteration of the key/value pairs. By merging the invalidated set into another, we could end up either not iterating it or iterating it twice.

The approach we propose, therefore, is to reuse our work from earlier and use a specialised indexing map that stores its keys in a PersistentWeakRef (*cf.* Section 4). Recall that only one such weak reference is constructed for each runtime value, and that once that value expires, calling **get**() returns the weak reference itself. The result is that the indexing map is still fully iterable after the key expires, and key lookups are possible while the key is alive, which is what we aimed to achieve. It is easy to see that this does not result in additional space leaks, since after a key expires the memory overhead for having seen a runtime value is constant and very small, and will be fully eliminated once associated partial matches complete or fail.

5.3 Updating Indexed Disjuncts

We follow the code-generation policy previously described for tracematches [2]. A method is generated for each tracematch symbol. This method is triggered when an event occurs which matches that symbol, for some variable bindings. The method uses these bindings to calculate changes to the constraints labelling the automaton. These changes are temporarily queued. Once a method has been run for each symbol that matches the event, the queued changes are used to update the main constraints.

Without indexing, the pseudo-code for the method which updates the constraints for a symbol a is:

```
def update_a(event_bindings):
  for (j,i) in a-transitions:
    label[i].pos = label[i].pos or
                   (label[j].original and event_bindings)
  neg = neg and not(event_bindings)
```

Such a method is run for each symbol that corresponds to an event. After each applicable update method is run, the results are combined:

```
def combine():
  for i in states:
    label[i].original = label[i].pos or
                        (label[i].original and neg)
    label[i].pos = false
  neg = true
```

Note how the two halves of these methods correspond to the two lines of Equation 1, respectively. The only difference is that here the results are imperatively built symbol-by-symbol using temporary variables. The variables are as follows:

- label[i].original — a set of disjuncts storing the constraint for state i
- label[i].pos — a temporary set of disjuncts in which the positive updates for state i are accumulated
- neg — a temporary set of disjuncts in which the negative updates for all states are accumulated

To refine this approach to use indexed constraints, the first step is to replace the sets used for the 'original' sets with the multi-level maps discussed above. The 'pos' sets are replaced with linked-lists called 'qeueue'. Once the relevant disjuncts for an update are located in the multi-level map, the results of processing them are stored using the 'queue' lists. We use lists rather than sets, because the queues must be reset every iteration and this adds large amounts of overhead with sets.

The pseudo code for the update methods that use indexing is as follows, although note that what is shown here is just pseudo code for clarity; in the actual implementation, specialised code is generated for each operation shown here and the loops are statically unrolled.

```
def update_a(event_bindings):
  for (j,i) in a-transitions:
    for disjunct in label[j].original.lookup(event_bindings):
      label[i].queue.append(disjunct and event_bindings)
  neg.andNot(event_bindings)
```

```
def combine():
  for i in states:
    label[i].original.and(all_event_bindings)
    label[i].original.insertAll(label[i].queue)
    label[i].queue = []
  neg.reset()
```

Consider again the safe-enumeration example. In contrast to the five calculations shown in Figure 4, computing the same NEXT update, for $(e = e_2)$, now only two disjunct calculations — one positive and one negative update to $(v = v_1 \wedge e = e_2)$ — and four relatively inexpensive map lookups.

5.4 Evaluation

Let us now examine the effect on the benchmark times, as displayed in Table 4.

ID	ASPECTJ	FULLOPT	LEAKELIM	NOOPT
1	3.3s	15.3s	97s	>90M
2	0.6s	1.6s	61.5s	748s
3	478s	627s	>90M	>90M
4	2.7s	3.1s	3.9s	5.5s
5	4.4s	9.8s	9.4s	9.2s
6	2.9s	3.3s	3.3s	3.3s
7	18.0s	21.2s	21.4s	21.6s
8	8.0s	9.5s	11.4s	12.0s

Table 4. Complete benchmark results.

Not surprisingly, in some cases indexing *deteriorates* performance, in particular for OBSERVER (5). As we mentioned earlier, in this application each subject has exactly one observer. It follows that the indexing structure only adds to the overhead of accessing that one element. This type of slowdown could be eliminated by introducing indexing in a dynamic fashion, only building the index when the number of disjuncts in a set exceeds a given threshold.

Overall, however, the effect of indexing is hugely beneficial. In the case of NULLTRACK (2), it reduces the execution time from 61.5s to 1.6s, which is particularly impressive considering the huge number of instrumentation points. Furthermore, APROVE (3) now becomes feasible to execute, and we can observe a huge speed-up in SAFEENUM (1). HASHCODE/WEKA (4) and REWEAVE (8) also benefit.

Perhaps the most pleasing side-effect of indexing, however, is the elimination of the crucial dependency on garbage collector performance that we observed in Section 4: It is now the case that running a benchmark with more memory will not automatically mean worse performance, since we only iterate relevant disjuncts. Of course it is theoretically possible that we update disjuncts that are relevant but would have expired after a garbage-collector run, but we weren't able to measure this in practice.

We conclude that the combination of leak elimination and indexing is a *conditio sine qua non* for the generation of efficient trace monitors. Neither of the two techniques would work as well in isolation: Without indexing, leak elimination depends on the JVM's memory measurement, and without leak elimination, indexing would run out of memory on any substantial benchmarks.

6. RELATED WORK

In the introduction, we already indicated that while there is a substantial body of work on trace monitoring, there are not a lot of systems available. As the focus of this paper is efficient implementation, we only review such systems here. Our original intention was to provide a detailed comparative study of the most mature trace monitoring systems; it turned out, however, that many of the systems were not available to the general public, and even with those that were, we frequently ran into basic problems that prevented our experimental evaluation.

Table 5 gives an overview, comparing the salient features. The first five systems in the table are all publicly available and allow trace monitors to be applied to Java programs. They are, therefore, broadly comparable — even though in AspectJ event sequences must be matched by hand-coding the monitor. The five systems on the bottom of the table are either not publicly available, or (in the case of Arachne) apply to another programming language, namely C.

The table attempts a comparison with respect to a number of criteria. Firstly, the purpose of the system: Many are geared solely towards runtime verification, whereas others (mostly with a background in aspect-oriented programming) are actually intended to augment the monitored program by running extra code when a matching trace is found, or maybe by replacing an event with new code.

Next, we examine the issue of integration with a programming language. Several of the systems are deeply integrated with AspectJ, but some others (for instance PQL) are stand-alone tools. The advantage of programming language integration is enhanced checking of the specifications at compile-time.

There is considerable variety in the way patterns are specified. Not all systems allow variables to be bound by the matching process: without such binding, it is difficult to write patterns that monitor the behaviour of a specific set of objects. The 'exact-match' column in Table 5 refers to the matching process. There are two different styles of semantics: One can either demand that every single event be accounted for by the pattern, or one can allow arbitrary events to occur in between matched statements, as does, for example, PQL. We refer to the former as an 'exact-match semantics', and to the latter as a 'skipping semantics'. The precise implications of this design choice are very interesting, but beyond the scope of this document; we refer the interested reader to [6] for an in-depth discussion.

Finally, a number of systems allow context-free patterns as opposed to merely finite state machines. While we have not considered the implementation of such rich patterns in this paper, it is clear that the same techniques apply there to avoid space leaks, and to index partial solution sets.

The next section of Table 5 examines the characteristics of the implementation. Only very few systems have based their implementation on a semantics. For tracematches, a proof of the correspondence between its declarative and operational semantics is presented in [2]. Tracematches pioneered the use of leak prevention and indexing as described in this paper, though these techniques have been picked up by JavaMOP to some extent — see the relevant discussion below. Other systems are not concerned with space leaks, and pay the associated performance penalty. The authors of HAWK kindly agreed to run our SAFEENUM benchmark for us (HAWK is not available for download), but memory leaks proved prohibitive. Our experience with PQL is described below.

Tracematches are also the only system that automatically specialises the generated code to the pattern — again, of course, without using interprocedural analysis. Further drastic improvements in efficiency are possible in some applications when interprocedural analysis *is* employed. The most sophisticated system of this kind is PQL, employing a BDD-based static analysis to rule out instrumentation points at compile-time. Unfortunately, we were not able to get the static analysis to work. A similar optimisation has been tried in the context of tracematches [9]: The findings were not very encouraging, showing that often the static analysis made only a very small difference in runtime.

The final column indicates whether a system can be freely downloaded. Where this was the case and the system could process Java, we tried to express our benchmarks. The performance of J-LO on SAFEENUM was such that we gave up on attempting further experiments (with its author's blessing). Our experience with AspectJ has been presented in this paper as the hand-optimised gold standard against which other systems should be measured. The time spent coming up with implementations in each case was substantial. The findings with the remaining two systems were more interesting.

PQL The Program Query Language (PQL [25]) was proposed as a stand-alone tool to find bugs in Java programs by writing queries over execution traces. A PQL query can

SYSTEM	PURPOSE			PATTERNS			IMPLEMENTATION					
	fault finding	functionality	integration	variables	exact-match	context-free	semantics	leak busting	indexing	specialisation	static match	availability
tracematches [2]	$\pm$	+	+	+	+	-	+	+	+	+	+ [9]	+
PQL [25]	+	-	-	+	-	+	-	-	-	-	+	+
J-LO [26]	+	-	+	+	-	-	+	-	-	-	-	+
JavaMOP [11]	+	+	-	$\pm$	-	-	-	$\mp$	$\mp$	-	-	+
AspectJ [4]	-	+	+	-	-	-	-	-	-	-	-	+
tracecuts [30]	$\pm$	+	+	-	+	+	-	-	-	-	-	-
PTQL [21]	+	-	-	+	-	+	-	-	-	-	+	-
HAWK [14]	+	-	-	+	-	+	-	-	-	-	-	-
Alpha [8]	$\pm$	+	-	+	+	+	-	-	-	-	-	+
Arachne [19]	$\pm$	+	+	-	+	-	-	-	-	-	-	+

Table 5. Systems for trace monitoring.

be named, can make use of free variables, and picks out events by writing fragments of concrete Java syntax. It employs a *skipping semantics*, that is, it allows any event to occur between matched statements. Due to the fact that the named queries can be (mutually) recursive, PQL can express context-free properties of the trace quite naturally.

PQL does not include any optimisations to avoid space leaks, and indeed when we encoded SAFEENUM, we observed a steep linear growth of memory usage over time: it was impossible to complete the benchmark without providing the JVM with more memory (*cf.* Figure 8 for a comparison with MOP and tracematches). Still, PQL completed the benchmark in 580 seconds, significantly faster than the naïve tracematch — but also rather slower than tracematches with leak elimination (97s) or full optimisations (15.3s).

Unfortunately, we ran into significant problems when trying our other benchmarks. Several of them (both HASHCODE benchmarks, for example) cannot be expressed due to limitations of the PQL language; the problem with HASHCODE is that PQL cannot bind primitive types like **int** (we confirmed this with the authors of PQL). Also, it is impossible to intercept and bind assignments to fields, and so we couldn't express NULLTRACK or REWEAVE. DBPOOLING and LUINMETH are both expressible, but do not work with the PQL 0.1 or 0.2 implementations for technical reasons.

JavaMOP JavaMOP is an implementation of the paradigm of monitor-oriented programming [11]. It provides a framework for so-called logic plugins to generate a trace monitor from their own domain-specific trace pattern language; such a plugin for regular expressions is predefined, making it natural to compare our work on tracematches with JavaMOP and to investigate to what extent our findings carry over. However, several different plugins (*e.g.* for LTL) are also available, and many of the design decisions in the system are influenced by the need to keep the core plugin-independent. Indeed, one of the main design goals of JavaMOP is to stay as general as possible with respect to the MOP framework. It supports three kinds of monitors (inline, outline and offline), and can trigger extra code both when patterns are validated and violated.

In what follows, when we say "JavaMOP", we shall mean "JavaMOP with the regular expressions plugin, inline monitors and validation handlers", as this configuration is closest in spirit to tracematches.

The system generates AspectJ source code, which then needs to be compiled with an AspectJ compiler to produce the instrumented program. Our discussion of JavaMOP will be structured as follows. First, we discuss some subtle differences with tracematches at the level of language design, as they have some impact on the implementation. Next, we zoom in on the implementation of JavaMOP, comparing it to tracematches both in terms of space usage and time efficiency. To be precise, we shall compare against the so-called 'centralised indexing' implementation of JavaMOP — the alternative, 'decentralised indexing' is briefly reviewed in our Future Work section. Finally, we draw some conclusions from this detailed comparison between tracematches and JavaMOP.

Language design. Broadly speaking, JavaMOP is very similar to tracematches. However, there are four significant differences:

- First, all variables must be bound by the first symbol in a pattern. This condition is not satisfied by two of our benchmarks. In NULLTRACK, the pattern is '*setnull anyget npe*', and the *anyget* symbol binds information about the field read, reporting it when a match occurs. The condition that all variables be bound by the first symbol also fails in the LUINMETH benchmark.

- Second, at most one state in the automaton corresponding to the pattern can be associated with each set of variable bindings. In tracematches, no such restriction exists. This can be a problem, since typically the automata generated for trace properties are non-deterministic, and so only keeping one of all the different states such an NFA might be in will miss some matches. (One could, of course, determinise the pattern at the cost of an exponential blow-up, but JavaMOP does not do this.)
- Third, symbols (or "events", as they are called in JavaMOP) are considered atomic, and so there is no provision for overlap — if multiple different symbols match the same runtime event, the monitor state is updated once for each such symbol. This is a conscious design decision, but one which significantly simplifies the required monitor update code.

  In tracematches, the symbols in the pattern are simply logical properties, and so it is natural that they can overlap; the implementation takes special care to preserve the semantics in this case. Updating matching state several times for a single event and potentially triggering the tracematch multiple times with the same bindings after just one event would clearly run contrary to our semantics [2].
- Fourth, in JavaMOP, variable bindings from the pattern cannot be directly used in the code that is triggered when the pattern matches. This is in part a consequence of the logic-plugin-independent design of the system. If we wished to use the bindings, they must be explicitly maintained by hand within the plugin. By contrast, tracematches do not impose such a restriction, freeing the user from the need to maintain bindings manually.

The designers of JavaMOP have thus made a number of pragmatic decisions, balancing expressiveness against ease of implementation and their overarching goal of preserving plugin-independence. As described in [2], the design of tracematches takes a formal semantics as its starting point, and then implements exactly that semantics, without making any concessions to facilitate more efficient implementation. It is interesting, therefore, to examine the price one pays for such generality, and the benefits that derive from JavaMOPs design choices.

Implementation. Some of the optimisations and benchmarks proposed in this paper were first disseminated in a technical report, and pleasingly the developers of JavaMOP incorporated parts of our work into their system. At the time of writing, several of our benchmark trace monitors had been expressed in their formalism and made available on their web page [17]; also, a weak form of leak elimination and indexing is supported by JavaMOP in a plugin-independent way.

The main difference between our approach and that of JavaMOP is in the representation of partial matches: While tracematches conceptually store an automaton labelled with

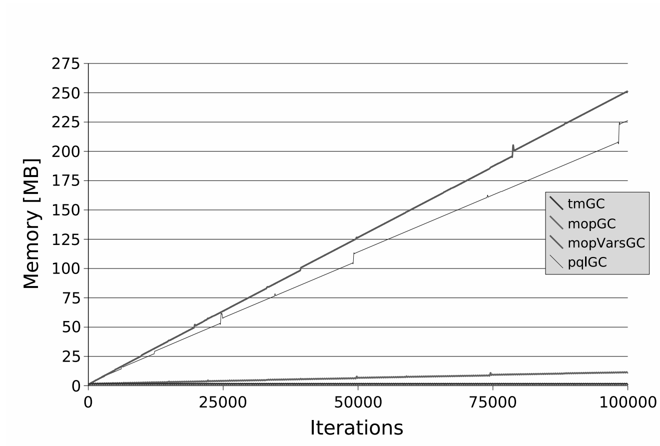

Figure 8. Memory usage for SafeEnum (top to bottom line): JavaMOP using bindings, PQL, JavaMOP not using bindings, tracematches using bindings.

constraints, JavaMOP uses a dedicated monitor instance for each distinct set of variable bindings, and updates a state counter on that. As mentioned above, this entails several problems, particularly the fact that it is impossible to use different space leak elimination strategies on different automaton states. Indeed, the approach taken by JavaMOP is almost disarming in its simplicity: Store each monitor instance in a set of indexing trees, one such tree for each tuple of variables bound by some monitor event, in such a way that keys are allowed to expire, and when they do associated sub-trees are discarded. In this way, monitor instances are kept alive for just as long as some event enabling them might occur (note, however, that this is not as effective as our collectable-binding-sets, or even just **collectableWeakRefs**, because if there is some event that binds no monitor variable, all monitor instances would be stored in a set without ever being removed).

Unfortunately, this scheme breaks down as soon as the user needs to inspect the acquired variable bindings and use them upon a successful match. According to the JavaMOP developers, the proper procedure there is to instrument the events with additional code that stores the binding in fields of the monitor instance, and then use those fields. Unfortunately, this implies keeping strong references to bindings, which will prevent them from ever being cleared.

Moreover, it is not possible for the user to avoid space leaks by manually using weak references when storing the bindings in fields, since that wouldn't guarantee their availability at the time of match completion. The only way around this seems to be an automaton-state specific handling of bindings, as implemented in tracematches. However, such an advanced strategy seems infeasible without in-depth knowledge of the underlying formalism, and so this cannot be implemented in JavaMOP without sacrificing some of the plugin-independence.

Figure 8 illustrates the effectiveness of space leak elimination on the SAFEENUM benchmark for tracematches and MOP. We can see that the sophisticated leak elimination strategy described in Section 4 is successful, and the memory usage for tracematches is essentially constant, even though all bindings are used in the body (the line is almost superimposed on the X axis, at an average memory usage of 1.1MB). The second line from the bottom corresponds to JavaMOP with a validation handler that doesn't use the bindings. Memory usage is reasonable, but there is still a clear upwards trend. The next line corresponds to PQL, and the top line corresponds to storing bindings in fields of the monitor instance and using them in the validation handler, as advised by the developers of JavaMOP. Essentially, this results in no heap object that was bound ever being released for garbage collection, and memory usage explodes.

It is also interesting to compare the performance of the two systems, since typically JavaMOP's approach of storing an automaton instance for each set of bindings seems more natural than the alternative of annotating states with constraints. As we observed above, it forces the restriction that any free variable of the monitor must be bound in the first observed event, and we wanted to determine if the price that tracematches pay for their generality is prohibitive.

Therefore, we tried to express all of our benchmarks in JavaMOP, guided by the examples on its website [17]. As we have already observed, NULLTRACK and LUINMETH violate JavaMOP's assumption that all variables be bound upfront, so these cannot be expressed directly.

Still, it is sometimes possible to manually tweak such cases into the form expected by JavaMOP, by rephrasing events or the entire pattern, or perhaps by using so-called *event actions* — these are blocks of Java code that run when an event matches, and while they somewhat undermine the declarative nature of the specification, they certainly increase expressiveness. We were able to express NULLTRACK using such event actions — the location and line number to be reported is stored on fields of the monitor instance and destructively updated with each new event match. Note that this technique would *not* work in cases which require different monitor instances for different values of the binding that is being handled in this way, but NULLTRACK happens to fit the bill. It is important to realise that this actually significantly simplifies the pattern from JavaMOP's point of view, since it reduces the number of variable bindings it needs to keep track of and index on. No such assumption is available to the tracematch.

The above technique doesn't work for LUINMETH, as different monitor instances for each binding *are* needed. One might think it possible to emulate it manually in event actions, but there is currently no way in JavaMOP to inspect the call stack, and so this benchmark is not expressible at this time (we have confirmed this with the designers of JavaMOP).

Finally, *DBPooling* requires *around* symbols (*i.e.* symbols that intercept and replace the last matching event), which MOP does not support. In summary, six of our eight benchmarks are expressible in JavaMOP.

The overall results are shown in Table 6. The MOP number for SAFEENUM shown is for the monitor that doesn't use the bindings in the validation handler (and hence almost succeeds in eliminating memory leaks), as that was significantly faster. As we can see, JavaMOP outperforms tracematches on HASHCODE/APROVE, OBSERVER and REWEAVE, typically by a margin of 20% or less, while tracematches hold the upper hand on the SAFEENUM (due to the superior leak elimination) and HASHCODE/WEKA. The NULLTRACK numbers in the table are not directly comparable, as the tracematch does more work: JavaMOP avoids the need to index on the source location and line number by use of event actions, as described above.

Conclusions. In conclusion, JavaMOP highlights some interesting restrictions to the language design that enable simpler code generation than that for tracematches. For some benchmarks, the savings can be as high as 20%, but the price paid is that many patterns are not expressible. It is natural, therefore, to wonder whether those same restrictions could be implemented as optimisations for tracematches.

The primary reason for the efficiency savings is JavaMOP's built-in requirement that all monitor variables be bound by the first observed event, which allows for more straightforward update code (in the terms used in Section 5, the negative updates never modify partial matches, they either leave them unchanged or discard them). We intend to implement a similar optimisation in the tracematches system on an on-demand basis — because we generate specific update code for each automaton state, it would be easy to use the above optimisation at *any* state that guarantees all variables bound, which, in the special case that the first event binds all variables, would give us the same result as introducing the restriction from JavaMOP.

Another opportunity is to exploit non-overlapping symbols. As said, in JavaMOP events are always unique, and no two symbols can match the same event simultaneously — some arbitrary order is imposed. In tracematches, we could implement an analysis that *proves* disjointness of symbols. Using such disjointness information, it is possible to make destructive updates to the constraints that label states, and that in turn would lead to the generation of simpler (and hence more efficient) code.

Static analyses In the introduction we already alluded to techniques that have been devised for static type-state verification (*e.g.* [18]). These analyses can be employed to show that certain program points can never contribute to the successful match of a particular trace pattern, thus avoiding the need for instrumentation — indeed, a staged analysis following this approach and extending it to the context of tracematches was recently proposed in [9, 10].

ID	MONITOR	BASE	KSLOC	TM	MOP
1	SAFEENUM	JHOTDRAW	9.5	15.3s	19.3s
2	NULLTRACK	CERTREVSIM	1.4	1.6s	0.8s
3	HASHCODE	APROVE	438.7	627s	590s
4	HASHCODE	WEKA	9.9	3.1s	3.6s
5	OBSERVER	AJHOTDRAW	21.1	9.8s	10.3s
8	REWEAVE	ABC	51.2	9.5s	8.3s

Table 6. Comparison of benchmark runtimes between tracematches and JavaMOP

The analyses of [9, 10] are particularly effective when tracematches are blindly applied to a large number of programs, because then tracematches fail to apply for easy-to-check reasons (say, no locks are acquired or released at all, so LUINMETH fails to apply overall). Indeed the numbers reported in [9,10] are very encouraging for that type of tracematch usage.

The situation is somewhat different, however, when benchmarks actually exercise all events in a tracematch, as is the case in our benchmark suite. For instance, one cannot hope to completely eliminate the instrumentation costs in examples like SAFEENUM, since it catches actual violations of the property in question. Moreover, such analyses suffer from the usual problems for whole-program and callgraph analysis: It is very hard to make sure the result is sound in the presence of multi-threading, dynamic class loading and reflection, which are quite common in real-world Java programs. Indeed, technical problems connected with these issues prevent us from reporting numbers for our benchmarks with the analysis from [10] applied.

However, it is important to realise when such static analyses will help. Typically, they are concerned with removing provably unnecessary instrumentation. When writing trace monitors for runtime verification concerns, a successful match indicates that the program misbehaved, and so in an ideal case one might hope that a static analysis could remove all instrumentation, proving the program correct with respect to the given property.

However, when viewing trace monitors as an extension of pointcuts in an aspect-oriented setting, they typically contribute an essential part of the system's functionality — our OBSERVER example is just such a case. Such monitors by definition can *not* be optimised away, because they will match. Particularly in this context, it is therefore important to do the best possible code generation for the given pattern, as there is no hope that a sufficiently sophisticated analysis might remove all overheads. The techniques presented in this paper are indispensable for that.

7. FUTURE WORK

It is natural to ask whether further improvements are possible, beyond the optimisations presented here. We are currently investigating several possibilities:

Reducing redundancy One obvious way to carry our indexing scheme further would be to note that it is redundant to store binding information both labelling the edges of an indexing tree, and on the partial matches at the leaves. We could then further specialise the partial match representation by discarding the redundant information. In the extreme case where every variable appears as an index, we would only have to store a simple counter recording the current automaton state.

Some care has to be taken, however — this is only well-defined if any symbol that can occur at the start of a matched trace binds all tracematch variables. In fact, this occurs frequently, and so such an optimisation seems promising: we already discussed this under JavaMOP in the Related Work section. Indeed, since tracematches already generate specialised code for each automaton state, one could take this further by using this technique for all variables for which it makes sense, for each state.

Disjoint pattern symbols Also, in our earlier discussion of JavaMOP, we noted that JavaMOP's simple code generation is partly due to its decision not to consider overlapping symbols in trace specifications.

Because tracematches are embedded in the AspectJ language, and there pointcuts can overlap, it is not possible to follow that design here. However, it is easy to envisage an analysis that proves disjointness of particular symbols. Using the result of such an analysis, we can simplify the code that updates constraint labels on automaton states for the disjoint symbols, making most operations destructive.

Bound variable correlation In many examples, there exists a many-to-one relationship between the objects bound in a trace monitor. For example, every enumeration corresponds to one collection, every observer has one subject, and so on. We can thus improve the implementation of indexing by moving the first level of the indexing tree into fields on objects. For example, in the OBSERVER tracematch, we might store all observers as a field on the subject. Note that this has the added benefit that when the subject is garbage collected, so are its observers.

Implementing this automatically requires some annotations on the specification, as well as a fairly complex analysis of the base program, however, going well beyond the cheap techniques we have introduced here. In particular, such analysis is necessary when it is not deemed acceptable to modify library code, to check that certain objects are constructed only in the compiled code, and not elsewhere. In the presence of such an analysis, we could generate code that is much closer to our hand-coded AspectJ gold standard for benchmarks like SAFEENUM, as described above.

A related optimisation was introduced by the designers of MOP in [11] for the special case of single-variable specifications only. In that context, the above would take the form of simply storing the monitor instance on a field of the bound object. We first sketched an extended technique incorporating handling of multiple variables in [2], and a limited version is implemented in JavaMOP [12, 17]. That implementation, called 'decentralised indexing', avoids the need for an expensive analysis by assuming it is permissible to transform the code of the Java standard libraries, and in many applications that assumption is not satisfied.

Dynamic indexing We noted that for some benchmarks, indexing aversely affects performance because the indexed sets are too small; in particular, this is the case for OBSERVER applied to AJHOTDRAW. The obvious solution is to set a threshold: sets below the threshold are not indexed, and those above it are.

A lot remains to be done to optimise trace monitor performance. It is clear, however, that we have identified the two essential techniques that make trace monitoring feasible in the first place.

8. CONCLUSIONS

This paper demonstrates, for the first time, how feasible trace monitors can be generated from specifications. It thus complements the substantial body of work that argued the desirability of trace monitors as a language feature.

This result was obtained through two techniques: the elimination of space leaks, and a sophisticated data structure for organising sets of partial matches. Neither of these techniques requires interprocedural analysis, and they can thus be employed without excessive compile-time costs. Our techniques approach the speed of hand-coded, hand-optimised monitors to within a factor of 3 at worst.

The leak elimination analysis was suggested in [2], but the strategy proposed there contained a crucial flaw that made it unsound. We have shown how to rectify this by introducing the novel notion of *persistent* weak references, and how to extend the results to effectively optimise a wider class of specifications. Furthermore, we presented a thorough experimental evaluation of the effectiveness of the new solution. The fact that the original flaw went undetected for so long is cause for some concern. At present there are no formal verification techniques available for data structures that make use of weak references. We are currently investigating the development of such verification techniques, using the data structure presented here as a motivating example.

All results of this paper, ranging from our benchmark suite to the optimisations, are applicable to most other trace monitoring systems, and we have thus opened the way for many comparative experiments in future. These are already starting to happen, as witnessed by the recent adoption of some of the techniques presented here by the JavaMOP system.

ACKNOWLEDGEMENTS

Neil Ongkingco did a huge amount of work on getting the benchmarks to run for an early version of this paper, in particular JigSaw and AJHotDraw. He also suggested many improvements to the paper itself — we're very grateful for his insights and his help.

We would also like to thank the other members of the *abc* team for their continued help and support.

We had inspiring discussions with the authors of JavaMOP, Feng Chen and Grigore Roşu, which greatly clarified the relation between JavaMOP and tracematches.

References

[1] The *abc* team. Benchmarks for Trace Monitoring. Scripts and sources to compile and run the benchmarks: `http://aspectbench.org/benchmarks`.

[2] 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 *Object-Oriented Programming, Systems, Languages and Applications*, pages 345–364. ACM Press, 2005.

[3] AProVE. Automated Program Verification Environment. `http://aprove.informatik.rwth-aachen.de/`, 2006.

[4] AspectJ Eclipse Home. The AspectJ home page. `http://eclipse.org/aspectj/`, 2003.

[5] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *Programming Language Design and Implementation (PLDI)*, pages 117–128. ACM Press, 2005.

[6] Pavel Avgustinov, Oege de Moor, and Julian Tibble. On the semantics of trace monitoring patterns. In *Runtime Verification*, 2007.

[7] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 04)*, volume 2937, pages 44–57. Lecture Notes in Computer Science, 2003.

[8] Christoph Bockisch, Mira Mezini, and Klaus Ostermann. Quantifying over dynamic properties of program execution. In *2nd Dynamic Aspects Workshop (DAW05)*, Technical Report 05.01, pages 71–75. Research Institute for Advanced Computer Science, 2005.

[9] Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Proceedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, page to appear. Springer, 2007.

[10] Eric Bodden, Patrick Lam, and Laurie Hendren. Flow-sensitive static optimizations for runtime monitors. Technical Report abc-2007-3, *abc* project, 2007. `http://abc.comlab.ox.ac.uk/techreports#abc-2007-3`.

[11] Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Workshop on Runtime Verification (RV'03)*, volume 89(2) of *ENTCS*, pages 108 – 127, 2003.

[12] Feng Chen and Grigore Roşu. Mop: An efficient and generic runtime verification framework. In David Bacon, editor, *Proceedings of OOPSLA 2007*, 2007.

[13] María Augustina Cibrán and Bart Verheecke. Dynamic business rules for web service composition. In *2nd Dynamic Aspects Workshop (DAW05)*, pages 13–18, 2005.

[14] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7. ACM Press, 2005.

[15] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura, and Mario Südholt. An expressive aspect language for system applications with arachne. In *Aspect-Oriented Software Development*, pages 27–38. ACM Press, 2005.

[16] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186. Springer, 2001.

[17] Grigore Rosu *et al.* JavaMOP homepage. `http://fsl.cs.uiuc.edu/index.php/JavaMOP`, 2007.

[18] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 133–144, New York, NY, USA, 2006. ACM Press.

[19] Thomas Fritz, Marc Ségura, Mario Südholt, Egon Wuchner, and Jean-Marc Menaud. An application of dynamic AOP to medical image generation. In *2nd Dynamic Aspects Workshop (DAW05)*, Technical Report 05.01, pages 5–12. Research Institute for Advanced Computer Science, 2005.

[20] Erich Gamma. JHotDraw. Available from `http://sourceforge.net/projects/jhotdraw`, 2004.

[21] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 385–402. ACM Press, 2005.

[22] Peter Hui and James Riely. Temporal aspects as security automata. In *Foundations of Aspect-Oriented Languages (FOAL 2006), Workshop at AOSD 2006*, Technical Report #06-01, pages 19–28. Iowa State University, 2006.

[23] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *European Conference on Object-oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[24] Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.

[25] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 365–383. ACM Press, 2005.

[26] Volker Stolz and Eric Bodden. Temporal Assertions using AspectJ. In *Electronic Notes in Theoretical Computer Science*, volume 144, pages 109–124, 2006.

[27] Arie van Deursen, Leon Moonen, and Marius Marin. AJHotDraw. `http://sourceforge.net/projects/ajhotdraw/`, 2006.

[28] Wim Vanderperren, Davy Suvé, María Augustina Cibrán, and Bruno De Fraine. Stateful aspects in JAsCo. In *Software Composition: 4th International Workshop*, volume 3628 of *Lecture Notes in Computer Science*. Springer, 2005.

[29] w3c. Jigsaw. `http://www.w3.org/Jigsaw/`, 2006.

[30] 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.

[31] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java implementations*. Morgan Kaufmann Publishers, 2000.