More efficient temporal pointcuts through dynamic advice deployment

Eric Bodden
Sable Research Group
McGill University
Montréal, Québec, Canada
ericbodden@acm.org

Volker Stolz
MOVES: Software Modeling and Verification
RWTH Aachen University
Aachen, Germany
stolz@i2.informatik.rwth-aachen.de

ABSTRACT

In previous work we and others have studied the applicability of various trace based matching approaches, such as tracematches [3], tracecuts [14] and DLTL [13, 5] (through our prototype tool J-LO). Such formalims provide users with an expressive matching language that gives explicit and well-defined access to an application's execution history. In some approaches, even free variables in expressions can dynamically be bound to objects on the execution trace. This keeps the burden of manual bookkeeping of state from the user.

In this work we demonstrate that besides the aforementioned issues of more convenient programming, such temporal pointcuts yield a large potential for possible optimizations through runtime deployment of aspects, due to their well-defined structure. Functionally equivalent code in pure AspectJ would not necessarily yield such a potential. This feature of trace languages adds well to static optimizations such as control flow and dataflow analysis as it has been proposed in [3]. We do not want to give a fully fledged end-to-end solution here, which would maybe restrict us to a certain specification formalism or runtime weaving approach. Instead, we show up general potential for optimizations through dynamic deployment as a pointer for future research on the field.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Interpreters*

General Terms

Languages, Performance, Verification, Theory

Keywords

Program monitoring, aspect-oriented programming, dynamic deployment

1. INTRODUCTION

In the previous months, a lot of effort has been put into trace based matching languages, which allow users to match not only based on a *current joinpoint* but also based on information of the *execution history*. Most of those mechanisms are language extensions of AspectJ and hence we want to concentrate on those. However, we believe that most other applications which use a formalism for temporal reasoning at runtime could benefit from the insights we provide. In particular, previous workshops in the field of Runtime Verification [1] have shown a growing interest by people of the verification community in aspect-oriented techniques. This is is due to the fact that despite the enourmous contributions which have been made to static verification approaches as Model Checking, some properties can still only be checked during the runtime of an application under test. Any hybrid technique making use of such checker components at runtime must of course induce *some* computational burden. The contribution of this work is to point into directions which promise potential for keeping this computational burden as low as possible.

In the following we give examples for trace based matching in different formalisms and compare to manual trace matching in naive AspectJ. We then briefly describe how trace matching formalisms are usually implemented and what implementation properties they share. This enables us to identify common potential for optimizations through runtime matching. We show that the explicit history access the formalisms provide is crucial to this optimization potential: An equivalent aspect in native AspectJ could not easily be optimized in the same way - the optimizations only become possible trough the explicit model provided by those formalism.

2. TRACE MATCHING APPROACHES

In the following we briefly introduce three approaches to trace matching, based on different formalisms namely regular expressions, linear temporal logic and context free grammars.

2.1 Tracematches

In 2005, Allan et al. introduced tracematches [3], an AspectJ language extension for history based matching based on *regular expressions*. The input to a tracematch is a strict sequence of named *symbols*, each such symbol essentially being a pointcut and a before/after specification. The tracematch is then formed as a regular expression over an alphabet defined by those symbols.

DAW'06 March 20th, 2006, Bonn, Germany.
ACM X-XXXXX-XXX-X/XX/XXXX.

```
1  tracematch(Vector c, Enumeration e) {
2      sym create after returning(e): call(Enumeration+.new(..)) && args(c);
3      sym next before:               call(Object Enumeration.nextElement()) && target(e);
4      sym update after:              vector_update() && target(c);
5
6      create next* update+ next
7      {
8              throw new ConcurrentModificationException();
9      }
10 }
```

Figure 1: Safe enumeration tracematch

The listing in figure 1 shows a tracematch detecting unsafe use of enumerations: Whenever an enumerator e is claimed for a vector c (event `create`), the vector is not to be updated (event `update`) as long as this enumeration is in use (i.e. `nextElement` is being called). Line 1 declares free variables *ds* and *e*, which *each possible* matching instance will be bound to. Lines 2-4 declare the symbol alphabet, while line 6 holds the regular expression in question: Enumeration creation (possibly followed by some calls to `next`) follwed by an update of the vector and some call to `next` should trigger an error message.

As one would guess, there might be properties, which may be cumbersome to express with regular expressions, especially safety properties, which state that *something bad should never happen*. Such a statement makes use of some implicit negation, which in regular expressions can only be implemented by converting the property to a regular expression *enumerating* the language of all possible path violating this property.

2.2 J-LO and DLTL

Hence our previous work, which was inspired by static and dynamic *verification* tools tried to overcome this drawback by providing negation and conjunction explicitly, introducing another specification formalism over pointcuts based on linear temporal logic (LTL), called dynamic LTL (DLTL). It is dynamic, because it features dynamic bindings of free variables in a similar way as the aforementioned tracematches. Besides, it is equal to usual LTL [11] with temporal operators **X** (next), **F** (finally), **G** (globally), **U** (until) and **R** (release). This logic can then be used to specify temporal assertions, which have to hold during each execution of the given application. Any violation of such a trace condition is reported at runtime. Our research prototype J-LO implements DLTL, currently in a purely dynamic way.

Table 2 gives an example specification stating that always (**G**) after a collection c has been added to a hash set s, this collection should not be modified on the subsequent path, unless it was removed from the hast set again.

2.3 Context-free patterns

There are other approaches around (e.g. tracecuts [14], PQL [9]) which allow for even more expressiveness, in particular for context-free patterns. We do not want to explain those in depth here, because handling of such features would lead us out of the scope of this paper. Section 4.5 however explains the rough idea of how our observations could possibly be of similar use to such formalism.

3. COMMON ASPECTS OF TRACE MATCHING

The aforementioned approaches, despite providing quite different specification formalisms, share quite some implementation details. The implementation of tracematches and J-LO for example both use finite state machines to propagate state over time. An implementation allowing for the specification of context-free expressions however might need to employ a pushdown automaton (stack machine) depending on the kind of pattern. Yet, all those automata, irrespective the fact of being finite or infinite are *triggered in the same way* - through declared symbols (pointcuts) in AspectJ: A transition is only triggered at well-defined join points, which need to be exposed by the AOP runtime that is employed.

And this fact yields optimization potential through dynamic aspect deployment: Maybe the interested reader has noticed in our examples already, that not all such symbols may actually trigger a state transition at any time.

Take for example tracematch about "safe enumeration": Here, between `create_enum` and `update_source` it does not matter whether there are zero, one or multiple occurrances of `call_next` — the pattern would match either way. We say that `call_next` is irrelevant in this state.

Or take the DLTL formula checking for safe use of hash sets: It does not matter to us whether a collection is actually modified or removed from a hash set unless it was added to a hash set before. Hence, our conclusion is that not every declared symbol is of interest at any time. Consequently, in those states the associated joinpoints should not even trigger an event in order to achive an improved performance. This can and should be achived by dynamic advice enablement.

But how can we algorithmically determine the set of relevant or irrelevant symbols? We give initial pointers for answering this question in the next section.

4. DETERMINING THE SYMBOLS OF INTEREST

Symbols of interest are those symbols which are able to change the internal state of a trace matching automaton — whatever this automaton might look like. By *state* we here refer to the full configuration of the automaton, i.e. in the case of stack-based automata, the stack content has to be taken into account for those considerations.

In the following we will however first explain the easier case where *finite* state machines are involved. This covers the implementations of tracematches and J-LO and might

```
1  Collection c, HashSet s:
2 G(
3   (
4     exit(call(* HashSet+.add(..)) && target(s) && args(c))
5   ) -> (
6     (
7       entry(call(* HashSet+.remove(..)) && target(s) && args(c))
8     ) R (
9       !(
10        entry(
11          (
12            call(* Collection+.add*(..)) ||
13            call(* Collection+.remove*(..)) ||
14            call(* Collection+.clear())
15          ) && target(c)
16 )))))
```

Figure 2: DLTL formula ensuring safe use of hash sets

also apply to special cases of more expressive implementations. We will cover such systems a bit deeper in section 4.5.

4.1 Regular expressions

The use of regular expressions leads naturally to finite state machines, which are usually deterministic but sometimes also determinized on-the-fly for enhanced efficiency. Here the notion of an irrelevant transition is easy as we can see in our example tracematch.

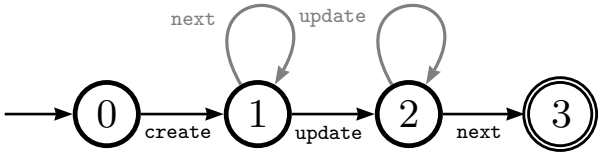

Figure 4.1 shows the (nondeterministic) state machine for the "safe enumeration" tracematch. (We abstract from *skip* transitions as they are mentioned in [3] because they do not contribute to, nor interfere with, the analysis we describe here.) Taking into account that in tracematches all events which are not part of the alphabet are ignored, obviously all loops (here shown in grey) can safely be dropped, still yielding the same semantics. For example, in state 1, it would make no difference if there occured a call to the `next` method of the associated enumeration.

Consequently, the associated joinpoints can be dynamically deactivated when such a state is reached. This leaves us for this example with the following *relevant* symbols for each state:

state	relevant symbols
0	{`create`}
1	{`update`}
2	{`next`}
3	$\emptyset$

In the following, we will denote this relationship by the function $rel : Q \to 2^{\Sigma}$ which returns for each state the set of relevant input symbols. Formally, we define

$$rel(q) := \{s \in \Sigma \mid \exists q' \in Q \,.\, (q, s, q') \in \Delta\}.$$

4.2 Dynamic deployment

Based on this information, we can now augment our original automaton with generic dynamic deployment commands:

- `deploy(`s`)` - Deploy advice associated to $s \in \Sigma$ so that symbol s can be triggered.
- `undeploy(`s`)` - Undeploy advice for s respectively.

We define the set of all such commands as:

$$C_{\Sigma} := \{\texttt{deploy(}s\texttt{)}, \texttt{undeploy(}s\texttt{)} \mid s \in \Sigma\}$$

This yields an enriched finite state machine with a labelling function $comm : Q \to 2^{C_{\Sigma}}$, which associates each state $q \in Q$ with the set of necessary deployment commands: Let q_p be the previous state observed, then in each state we deploy advice for those symbols which are relevant but have not been relevant before and we undeploy those symbols which have been relevant in q_p but are now not relevant any more.

$$\begin{aligned} comm(q) &= \{\texttt{deploy(}s\texttt{)} | s \in rel(q) \wedge s \notin rel(q_p)\} \\ &\cup \{\texttt{undeploy(}s\texttt{)} | s \notin rel(q) \wedge s \in rel(q_p)\} \end{aligned}$$

Note that this procedure may lead to relatively frequent (un)deployment. This might not be a problem, especially in environments, where such (un)deployment can be performed reasonably fast. However, depending on the overhead this (un)deployment may induce, one might want to use a larger window instead, i.e. one would undeploy a symbol only if it is irrelevant and has been so for the last n states. The exact parameters will of course heavily depend on the infrastructure in operation. Hence we leave this as open work for other researchers. For our example from above, the command function would look at follows:

state q	deployment command $comm(q)$
0	{`deploy(create)`}
1	{`undeploy(create)`,`deploy(update)`}
2	{`undeploy(update)`,`deploy(next)`}
3	{`undeploy(next)`}

Those commands are obviously to be applied immediately upon arrival at q.

4.3 The case of temporal logic

In temporal logic, it is usually even quite more often the case that certain events are ignored (as e.g. noted in [3, 5]). The LTL formula $\mathbf{G}(p \rightarrow \mathbf{F}q)$ requires *one* event q to follow whenever p is seen — it specifies nothing at all about events that could happen in between. Consequently, any path where always p is followed by q would satisfy the formula.

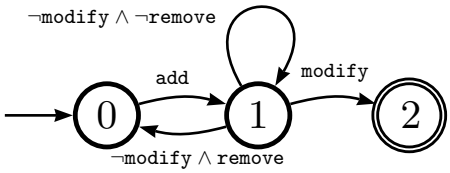

Figure 4.3 shows a finite state machine implementing the check for our hash set example formula (cf. figure 2). In contrast to regular expressions as they are used in tracematches, LTL is *propositional* i.e. it can distinguish between different propositions holding *at the same time*. Hence, the input to an LTL formula (or the equivalent automaton) is usually a sequence of *sets of symbols*, i.e. an element of $(2^{\Sigma})^*$. This is reflected by transitions with conditions as ¬`modify`∧`remove`: This transition is taken when `remove` matches the current joinpoint, but `modify` does not.

As we can however easily see, this change in the automaton model does not really affect our previous observations. Still we can determine the set of relevant symbols for any state of the automaton, just as before, now joining over all symbols contributing to a transition. (As for regular expressions, we assume that self-loops have already been eliminated.) Also the command function remains unaffected.

In our hash set example, this would yield the following result:

state q	$rel(q)$	$comm(q)$
0	{`add`}	{`deploy(add)`}
1	{`modify`, `remove`}	{`undeploy(add)`, `deploy(modify)`, `deploy(remove)`}
2	$\emptyset$	{`undeploy(modify)`, `undeploy(remove)`}

4.4 Per-object deployment

What we did not consider so far is one hidden peculiarity which is common to all the aforementioned formalisms: They all allow to *dynamically bind objects* in their specification. In other words, taking our LTL formula into account, the structure of the formula is not just

$$\mathbf{G}(add \rightarrow (remove\ \mathbf{R}\ \neg modify))$$

but rather:

$$\forall h \forall c : \mathbf{G}(add(h,c) \rightarrow (remove(h,c)\ \mathbf{R}\ \neg modify(c)))$$

So in order to apply the above deployment correctly, one would have to modify the command function accordingly:

state q	$rel(q)$	$comm(q)$
0	{`add(h,c)`}	{`deploy(add,(h,c))`}
1	{`modify(c)`, `remove(h,c)`}	{`undeploy(add,(h,c))`, `deploy(modify,(c))`, `deploy(remove,(h,c))`}
2	$\emptyset$	{`undeploy(modify,(c))`, `undeploy(remove,(h,c))`}

We are currently aware of one application, namely Steamloom [2, 4], which is going to provide such a feature in the near future. Certainly such support would most sensibly have to be integrated into the JVM in use.

4.5 More expressiveness: Stacks and counters

When moving from regular expressions to more expressive formalisms there are usually two natural directions to take: One is the one of allowing for context-free expressions (i.e. context free grammars, CFGs), which might lead to the necessity of bookkeeping state on a stack during runtime. Tracecuts for example is one system with this property. The second direction formally corresponds Petri Nets and provides for counting properties, i.e. expressions of the form $a^n b^n c^n$ for symbols a, b, c and some arbitrary $n \in \mathbb{N}$.

Again, we don't want to go into detail here, but instead we want to point to the important fact that the *reachability* problem is for both formalism, CFGs and Petri Nets, statically decidable:

For CFGs one can construct the so-called *p-automaton* [**?**] in polynomial time. This automaton is an ordinary finite state machine and accepts the language of all configurations reachable from one given configuration. By the use of such an automaton, one can hence decide, which symbols might still be of use on subsequent computations. Similarly, for Petri Nets, reachability is decidable [10, 7] by a special form of enumerating successor configurations - however in at least exponential time.

4.6 Applicability to non-AOP approaches

As mentioned above, several tools exist, especially in the Runtime Verification community, which are not directly related to the use of aspects or to aspect-oriented programming at all. Instead they seek to verify certain program properties at runtime. Yet, such tools could naturally benefit from our observations in the very same way, as also those tools might be able to speed up themselves by dynamic (un)weaving of employed instrumentation. PQL [9], PTQL [12], HAWK [6], and EAGLE [8] are for instance some candidates which could benefit from such efforts.

5. THE CASE OF PURE ASPECTJ

The purpose of this section is to show that formalisms as the one above, which *explicitly* match on the execution history of an application, is essential to the analyses we provide in this work.

Assume again the example of assuring safe enumeration, as it was conducted using tracematches. Assume further, one would have tried to implement the same functionality in pure AspectJ. One would have had no other option than generating an aspect containing at least three pieces of advice (one each for `create`, `update` and `next`) and then using those pieces of advice to conduct state transitions within the aspect. Figure 3 shows an excerpt of some pseudo implementation as it would be necessary. As would guess, an analysis of such an aspect for any temporal properties is in the general case impossible. Since the temporal structure is now "flattened" into independent pieces of advice, every potential for analysis of the temporal behaviour is gone. Hence we argue that the abovementioned approaches for a formal, explicit specification of temporal properties are indeed *necessary* to allow for such analyses in the first place.

```
1  aspect SafeEnum {
2
3      State state = new State(0);
4
5      after(Vector c) returning(Enumeration e): call(Enumeration+.new(..)) && args(c) {
6          if(state.inState(0)) {
7                  state = new State(1,e,c);
8          }
9      }
10
11     //two more pieces of advice here
12 }
```

Figure 3: Safe enumeration tracematch

6. CONCLUSION

In this work we have shown an overview of how properties of temporal specification languages can be exploitet for the purpose of efficiency gains through dynamic advice deployment. Such history based matching languages, opposed to pure AspectJ, expose an explicit temporal structure in their pointcut structure in order to match event patterns in the execution history of a running application. This temporal structure could successfully be shown to be rich enough to allow for an improved runtime performance by temporarily, dynamically unweaving parts of the matching machinery.

We have shown that this approach is applicable regular expressions and linear temporal logic (LTL) and to the tools implementing pointcuts based on those formalisms, namely tracematches and J-LO. Furtheron, we described how the mechanism could possibly be extended to more expressive formalisms making use of stacks and counters.

By an example we demonstrated that through the attempt of simulating a temporal pointcut of any of such formalisms though pure AspectJ, one loses the explicit temporal strucure, thus giving away any possibility for the analyses we propose.

7. REFERENCES

[1] *1st, 2nd, 3rd, 4th and 5th CAV Workshops on Runtime Verification (RV'01 - RV'05)*, volume 55(2), 70(4), 89(2), 113, ? Elsevier Science, 2001, 2002, 2003, 2004, 2005.

[2] 11 2005. Personal communication with Michael Haupt, Darmstadt University, Germany.

[3] C. Allan, P. Avgustinov, A. Simon, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding trace matching to AspectJ (*submitted to OOPSLA'05*). abc Technical Report abc-2005-01, McGill University, 2004.

[4] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM Press.

[5] E. Bodden. J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen university, 11 2005.

[6] M. d'Amorim and K. Havelund. Event-based runtime verification of java programs. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[7] J. Esparza and M. Nielsen. Decidability issues for petri nets - a survey. *Bulletin of the European Association for Theoretical Computer Science*, 52:245–262, 1994.

[8] K. H. H. Barringer, A. Goldberg and K. Sen. Program monitoring with ltl in eagle. In *18th International Parallel and Distributed Processing Symposium, Parallel and Distributed Systems: Testing and Debugging - PADTAD'04*. IEEE Computer Society Press, Apr. 2004. ISBN 0769521320.

[9] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. *SIGPLAN Not.*, 40(10):365–383, 2005.

[10] E. W. Mayr. An algorithm for the general petri net reachability problem. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 238–246, New York, NY, USA, 1981. ACM Press.

[11] A. Pnueli. The Temporal Logics of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 1977.

[12] R. O. Simon Goldsmith and A. Aiken. Light-weight instrumentation from relational queries over program traces. Technical Report UCB/CSD-04-1315, EECS Department, University of California, Berkeley, 2004.

[13] V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *Fifth Workshop on Runtime Verification (RV'05)*. To be published in ENTCS, Elsevier, 2005.

[14] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *SIGSOFT FSE*, pages 159–169, 2004.